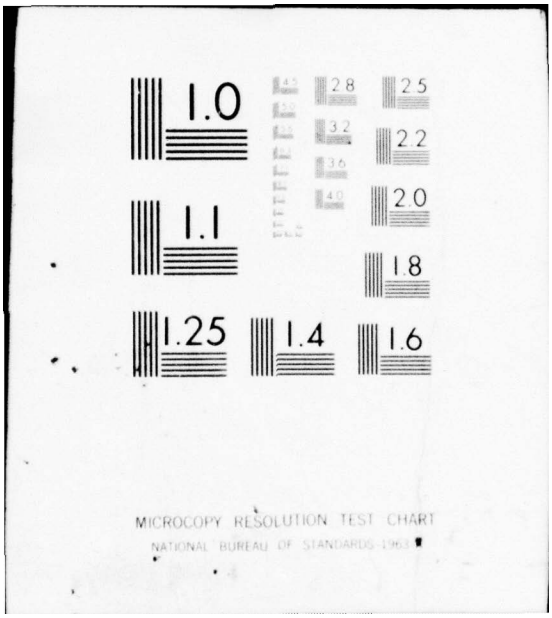
AD-A031 431 ILLINOIS UNIV AT URBANA-CHAMPAIGN COORDINATED SCIENCE LAB F/G 9/2
AN INTEGRATED CIRCUIT FAULT MODEL FOR DIGITAL SYSTEMS.(U)
SEP 76 M L KETELSEN DAAB07-72-C-0259
UNCLASSIFIED R-743 NL

1 OF 2
AD
A031431


1.0
1.1
1.25
1.4
1.6
2.8
2.5
3.2
2.2
3.6
2.0
4.0
1.8
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963

FC

REPORT R-743 SEPTEMBER, 1976 UILU-ENG 76-2231

CSL COORDINATED SCIENCE LABORATORY

12

AN INTEGRATED CIRCUIT FAULT MODEL FOR DIGITAL SYSTEMS

MARK LOREN KETELSEN

UNIVERSITY OF ILLINOIS – URBANA, ILLINOIS

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Date Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): AN INTEGRATED CIRCUIT FAULT MODEL FOR DIGITAL SYSTEMS,	5. TYPE OF REPORT & PERIOD COVERED: Technical Report
	6. PERFORMING ORG. REPORT NUMBER: R-743; UILU-ENG-76-2231
7. AUTHOR(s): Mark Loren Ketelsen	8. CONTRACT OR GRANT NUMBER(s): DAAB07-72-C-0259
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, Urbana, Illinois 61801	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 151 P.
11. CONTROLLING OFFICE NAME AND ADDRESS: Joint Services Electronics Program	12. REPORT DATE: September, 1976
	13. NUMBER OF PAGES: 143
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Doctoral thesis,	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Fault Detection
Fault Tolerance
Fault Model

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A model for the faulty behavior of digital networks realized using integrated circuit devices is proposed. This model, the pin fault assumption, is based on a study of the most frequently encountered failure mechanisms for such metworks, and the observation that previous fault assumptions model a large number of faults which occur with low frequency. The basis for the pin fault model, and an investigation of multiple fault detection using this model is presented.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED 097 700
SECURITY CLASSIFICATION OF THIS PAGE (When Date Entered)

DN

UILU-ENG 76-2231

AN INTEGRATED CIRCUIT FAULT MODEL FOR DIGITAL SYSTEMS

by

Mark Loren Ketelsen

This work was supported in part by the Joint Services Electronics Program (U.S. Army, U.S. Navy and U.S. Air Force) under Contract DAAB-07-72-C-0259.

AN INTEGRATED CIRCUIT FAULT MODEL FOR DIGITAL SYSTEMS

BY

MARK LOREN KETELSEN

B.S., Iowa State University, 1969
M.S., University of Illinois, 1973

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1976

Thesis Advisor: Professor Gernot Metze

Urbana, Illinois

AN INTEGRATED CIRCUIT FAULT MODEL FOR DIGITAL SYSTEMS

Mark Loren Ketelsen, Ph.D.
Coordinated Science Laboratory and
Department of Electrical Engineering
University of Illinois at Urbana-Champaign, 1976

A model for the faulty behavior of digital networks realized using integrated circuit devices is proposed. This model, the pin fault assumption, is based on a study of the most frequently encountered failure mechanisms for such networks, and the observation that previous fault assumptions model a large number of faults which occur with low frequency. The basis for the pin fault model, and an investigation of multiple fault detection using this model is presented.

Fault detection for combinational modules is investigated, and it is shown that multiple pin fault detection tests sets for such modules may be quite easily generated. The computation required to generate such test sets is independent of the circuit realization internal to the model, and each test generated requires about the same amount of computation. The computational complexity of test generation is greatly reduced compared to that for previously studied fault models.

Pin fault detection experiments for sequential machines are studied, and methods for designing such experiments are developed. These design methods are compared to those under other fault assumptions, and a substantial reduction is observed in the length of such sequences and the computation required to produce them.

ACKNOWLEDGMENTS

The author wishes to express deepest gratitude to his advisor, Professor Gernot Metze. Professor Metze's friendship as well as his guidance and encouragement have been unfailing throughout the author's graduate career.

The author would like to thank his colleagues in the Digital Systems Group, and particularly James Smith and Trevor Mudge. The technical support of the Coordinated Science Laboratory, and the clerical support of Mrs. Rose Harris are greatly appreciated.

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

1. INTRODUCTION

Reliability has been recognized as a primary engineering consideration in the development of digital systems for quite some time. Unlike most other electronic apparatus, failures in digital systems are in many cases not readily apparent. Consequently, the problem of ensuring the reliability of the computations produced by digital systems has been actively studied. This area, which has come to be known as fault-tolerant computing, has taken many directions, and has been studied from many application viewpoints.

1.1. Fault-Tolerant Computing

It has been recognized that reliability is not a problem which can be effectively dealt with after a system has been designed, but rather must be faced from the inception of the system design. The various approaches to the achievement of some degree of fault-tolerance may be loosely classified into two categories: fault diagnosis with repair, and fault masking.

The principle behind fault diagnosis and repair is that the system is tested either periodically or continuously, and upon the determination that a failure has occurred, computation is interrupted and the system is restored to failure-free status through repair. If this testing is done on a periodic basis, and a failure is detected during

some diagnostic interval, then all of the computation performed since the previous diagnostic interval is suspect. It therefore becomes necessary to repeat these suspected computations. If, on the other hand, the system has been designed so that diagnosis takes place concurrent with normal operation, failures are immediately detected, and the need to repeat a potentially large amount of computation is eliminated. The primary trade-off between these two approaches is that a significant amount of additional circuitry is required to effect on-line diagnosis, whereas periodic diagnosis requires little or no circuitry beyond that required to perform normal computation. An exception to the latter statement is the case where periodic diagnosis is an automated function of the system, and repair is achieved through switching from faulty subsystem modules to standby spares. The problem of generating a suitable set of diagnostic tests has received much attention [1,2,3]. The development of systems capable of currently testing themselves is a more recent development [4,5]. The possibilities for long range space exploration have prompted investigation into the feasibility and implementation of self testing systems with repair capability [6].

Fault-tolerance may also be achieved through the use of redundancy introduced in a way which prevents failures from propagating to a point where incorrect system operation would result. Redundancy which achieves this fault masking effect may take several forms. Triple modular redundancy [7] or the more general n-tuple modular redundancy [6] achieve fault tolerance through replication of functional modules, the outputs of which are compared, and the majority output propagated. Redundancy may

be introduced at the logic level by a technique known as quadded logic [8]. Automatic repair may also be employed in modular redundancy schemes by switching in standby spares to replace modules which produce minority outputs [6].

1.2. Fault Detection in Digital Systems

Regardless of the particular design approach used to achieve fault-tolerance, the ability to test the logic networks which implement the system is essential. For example, in the totally self checking approach [4,5], it is essential that the circuit be designed so that the input code space includes a test set for the circuit. In cases where modular redundancy is used to achieve fault masking, reliability enhancement occurs only if it can be guaranteed that initially all the hardware is functioning correctly. This requires a system checkout phase in which the system is reconfigured so that the replicated modules may be individually tested. Consequently, fault detection is a problem which is central to the whole realm of fault-tolerant computing.

The fault detection problem may be stated as follows. Given a digital network and a set of faults which can occur, derive a set of inputs which can be applied to the network such that normal response to the test set ensures that none of the faults in the fault set have occurred.

As this statement of the fault detection problem indicates, it is first necessary to define an appropriate set of faults. This set of faults represents an abstraction or model of the ways in which physical

failure phenomena effect the behavior of the network. The selection of a fault model determines the effectiveness of the solution to the fault detection problem, and is therefore a very important consideration. In addition, this selection also has direct impact upon the derivation and analysis of suitable test sets.

The selection of a fault model represents a trade off between completeness in terms of modeling all possible physical failures, and ease of generating test sets based on the model. Each physical failure mode has some probability of occurrence. The "coverage" of a fault model is the conditional probability that given a failure has occurred, the failure affects the network in the same way as one of the faults in the model. A good fault model is one which facilitates the analysis necessary to derive test sets, but also has an acceptably high coverage.

Existing fault models [9,10] are based on the assumptions that faults are "logical" and "permanent" [9]. The term "logical" means that failures transform the logical network into another logical network. This assumption is necessary in order to avoid having to analyze the voltage and current waveforms in the electronic circuits which realize the network. The term "permanent" means that once a fault has occurred that fault will remain until it is repaired, although other faults may occur subsequently. It is further assumed that failures may be modeled as permanent logical constants occurring at gate inputs and outputs. This fault model is known as the "stuck-line" fault model because the faulty network behaves as if certain gate inputs and outputs are "stuck" at a logical 0 or 1. This model is amenable to analysis because the gates which make up the network

retain their logical properties. The stuck-line model has proven to be quite effective in terms of fault coverage. Fault analysis under the stuck-line fault model is dramatically simplified if it is assumed that at most one gate input or output is stuck. This assumption is based on the reasoning that if the system is relatively reliable, the probability of more than one fault occurring between diagnosis intervals is acceptably low. However, this implies that at some point the network is fault-free, and further, that the network is diagnosed sufficiently often. Also it is implied that the physical failures are adequately modeled as single stuck faults. These assumptions are difficult to justify in many instances. More recently attempts have been made to relax the single fault assumption and consider multiple faults, i.e. faults made up of a set of gate inputs and outputs stuck at logical constants [11,3].

1.3. The Pin Fault Model

State-of-the-art digital system design techniques rely heavily on medium- and large-scale integrated circuit technologies which have been developed in the past decade. This continuing trend has allowed the implementation of digital systems of extremely high complexity. Existing fault detection techniques have become increasingly difficult to apply to such systems [12,13,14], since as gate counts increase, the single fault assumption becomes increasingly difficult to justify, while the number of multiple faults increases at an exponential rate.

This thesis proposes a new fault assumption, the pin fault model, which appears to retain an acceptable degree of fault coverage while allowing fault analysis and test generation to remain within the realm of computational feasibility. In this model, "stuck-at" type faults are assumed to occur only at input and output pins of integrated circuit modules. This substantial reduction in the number of fault sites, in addition to the implicit assumption that the logical network within the integrated circuit boundaries remains fault-free, allows multiple faults to be considered easily.

1.4. Outline of the Thesis

Chapter 2 introduces the pin fault model. Motivation for its application and justification in terms of fault coverage is presented.

In Chapter 3, fault detection under the pin fault assumption is examined for combinational circuits. Test sets for multiple pin fault detection are investigated, and a test set generation algorithm is developed.

Chapter 4 considers sequential circuits. The design of test sequences capable of detecting all multiple pin faults for synchronous and asynchronous circuits is studied.

Chapter 5 concludes the thesis. Design rules to enhance test generation under this fault model are discussed. Conclusions about the fault model are presented, and areas for further research are proposed.

2. THE PIN FAULT MODEL

2.1. Introduction

Medium- and large-scale digital integrated circuit devices make available single-package networks consisting of hundreds or even thousands of equivalent gates. The use of such devices in digital system design has become widespread because of the savings in cost, physical size, power consumption, speed, and reliability. As indicated in the previous chapter, the vast increase in the complexity allowed by such technologies has created a dilemma in the testing of such systems. This thesis proposes an alternative to existing testing methodologies which is based on a fault model that results in a significant reduction in the number of modeled faults.

2.2. The Proposed Model

A fault model is an assumption made about the behavior of networks which have suffered physical malfunction in the electronic circuits used to implement them. This assumed faulty behavior is the basis for analysis of faulty networks, and the generation of fault detection tests. Most previous work in the area of fault detection has been based on the stuck-line fault model. The fault model to be defined in this section retains the use of logical, permanent faults of the "stuck-at" type, however the set of possible fault sites is reduced.

In practice, integrated circuit modules which contain the necessary logical functions are interconnected to implement the desired network. The proposed fault model, called the pin fault model, assumes that stuck-at type faults occur only on the input and output pins of the integrated circuit modules of the network.

Figure 2.1 illustrates the fault sites for the stuck-line model for a carry-save full adder circuit which occupies half of an integrated circuit module. There are 33 such fault sites. Under the pin fault model, the set of fault sites consists of input pins C_n, B, and A, and output pins C_{n+1} and Σ. It can be seen that the reduction in the number of fault sites under the pin fault model compared to the stuck-line model becomes more pronounced as the ratio of internal lines to pins increases. Integrated circuit fabrication technology is limited by the number of chip leads which can be brought to the "outside world" (14-40 pins) rather than the amount of internal logic. As a consequence, the types of circuits which are integrated are typically those for which the pin fault model leads to a large reduction in the number of fault sites. This reduction in the number of fault sites has the potential for simplification of fault analysis and test generation, however it also has the potential for reducing the fault coverage. This implication is considered in the next section.

The pin fault model assumes that multiple as well as single stuck-at faults may occur on module pins. The desirability of the multiple fault approach has long been recognized, however the difficulties encountered in its application to the stuck-line fault model have been substantial.

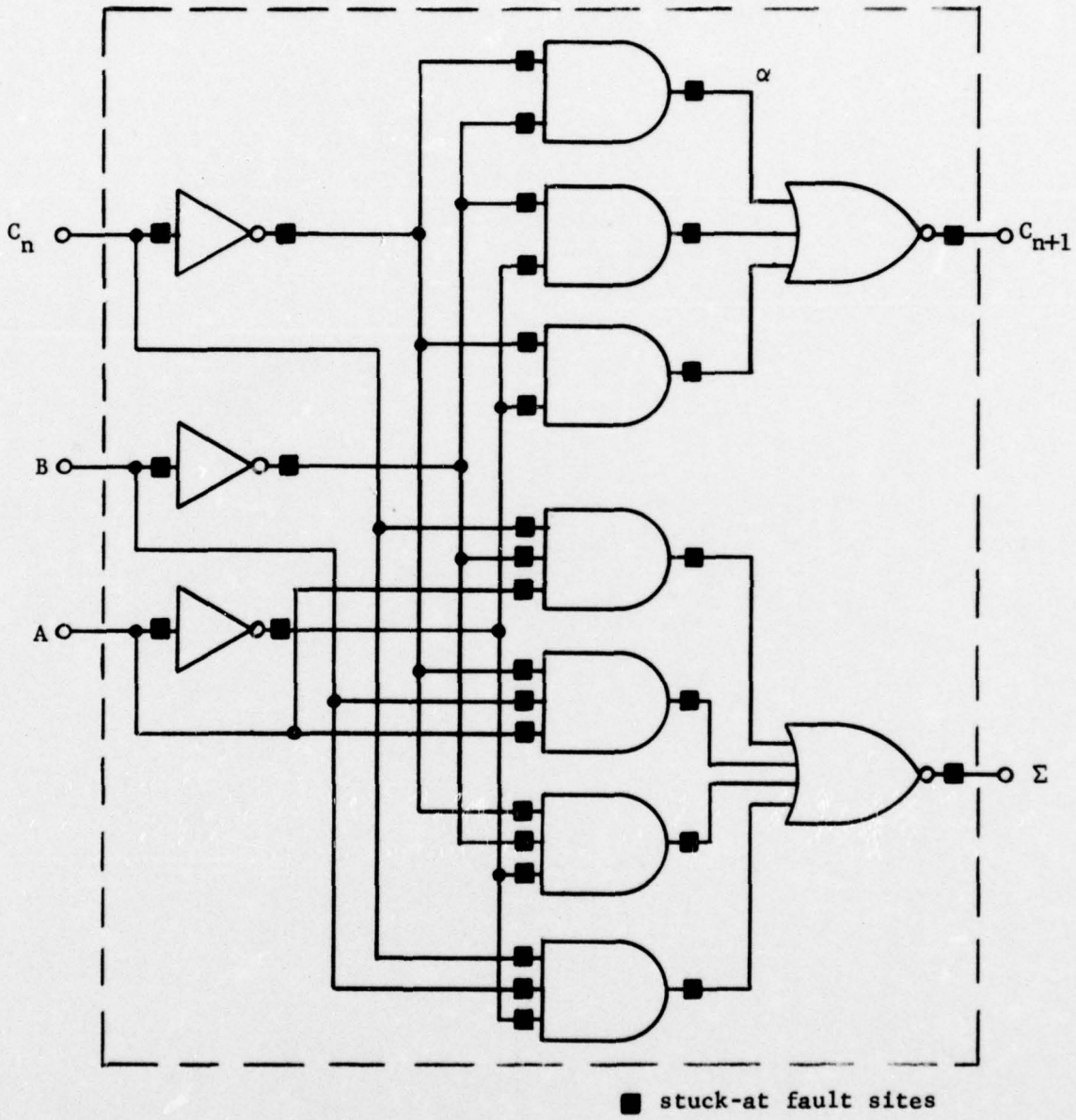


Figure 2.1. Carry-save full adder circuit with fault sites.

Subsequent chapters in this thesis show that the multiple fault approach is more tractable under the pin fault assumption. This multiple fault assumption greatly improves the fault coverage of the model. In addition to failure mechanisms which directly effect module pins, the multiple pin fault approach models many internal logical faults through fault equivalence.

2.3. Justification of the Pin Fault Model

In order for the pin fault approach to be valuable, it is necessary that it provide adequate fault coverage. The evaluation of this quality involves a statistical analysis of actual circuit failure modes in order to determine what fraction of all failures are modeled by multiple pin faults. The analysis of failure mechanisms is conducted primarily by semiconductor manufacturers, and data is not generally available.

Available failure data for digital integrated circuits show the dominant failure mechanism to be "lead bond" failure. These failures occur at the interconnection between the integrated circuit package pins and the silicon chip itself [15,16]. Such failures result from thermal and mechanical shock and high humidity. Failures may also result from electrical overload. For example, if the output pin of a TTL logic gate which is in the high state is inadvertently shorted to ground, as by an oscilloscope probe, the output transistors will fail. Another example is voltage "overshoot" on an input pin. Such electrical stress typically results in a failure of input or output transistors.

Overall system reliability depends on the reliability of the interconnections between integrated circuits and the distribution of supply voltages, in addition to the integrated circuit reliability. Present day system packaging techniques utilize multi-layer printed circuit boards to mount and interconnect the integrated circuit modules. The interconnecting lines are deposited on the boards by photolithographic techniques, and the integrated circuit modules are then soldered in place. These solder joints must make connections between the integrated circuit pins and the printed wires on interior layers. Studies have shown that these solder joints and the printed circuit routing are typically less reliable than the integrated circuit devices used in the system [16]. Therefore such failures represent a dominant failure mode for the overall system.

The common logic families (TTL, ECL, MOS) are designed so that, under normal operating conditions, the various parameters of the transistors, diodes, and resistors are quite stable with respect to time. This is the reason for the outstanding reliability of such devices (50 - 100 failures per 10^9 hours [17]). Semiconductor manufacturers do extensive testing of semiconductor chips during fabrication to eliminate devices whose parameters do not fall within the design margins. These two facts suggest that failure of some component within a silicon chip is highly unlikely, especially compared to lead bond and solder joint failures.

All of the failure mechanisms discussed are suitably modeled by the pin fault assumption, since they all result in logical behavior characterized by input and output pins being stuck at logical constants. Based on available data, the pin fault model appears to provide excellent

fault coverage. This conclusion implies that although the classical stuck-line fault assumption models more faults, most of these faults occur with negligible frequency.

3. FAULT DETECTION IN COMBINATIONAL MODULES

3.1. Introduction

The most fundamental context for considering fault detection strategies is at the level of the single-output combinational module. Development of fault detection theory at this level is basic to all types of digital systems.

The following nomenclature will be used throughout the remaining chapters. X is an input vector consisting of ℓ coordinates designated by subscripted variables, e.g. x_i is the i-th coordinate of vector X.

Definition 3.1: Combinational function Z(X) is a nonvacuous function of X if there is no component x_i of $X = x_1x_2\ldots x_\ell$ such that

$$Z(x_1x_2\ldots x_{i-1}\ 0\ldots x_\ell) = Z(x_1x_2\ldots x_{i-1}\ 1\ldots x_\ell).$$

Except where noted, all combinational functions will be assumed to realize nonvacuous functions of X. Figure 3.1 represents a combinational module.

In this chapter the problem of constructing test sets for combinational modules is considered. A theory for such testing is studied, and a test generation algorithm is developed. The computational feasibility of this algorithm for large modules is investigated. Finally minimum length test sets and the extension to multiple-output functions are considered.

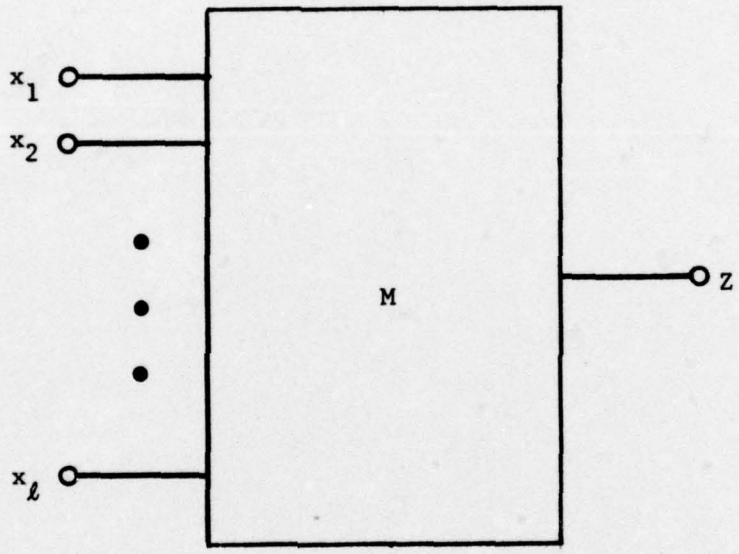


Figure 3.1. Module M realizing function Z = f(X).

3.2. Preliminary Discussion

The following definitions are presented to simplify discussion of the fault detection problem:

Definition 3.2: The multiple pin fault set, $\mathfrak{F}_M$, is the set of all possible assignments of stuck-at-0, stuck-at-1, and normal (fault-free) values, except the all-normal assignment, to the ℓ inputs and output of module M. The all-normal assignment is denoted as ϕ.

When considering realizations of functions which may contain faults, it is convenient to consider the output of the realization to be a function of both the input vector and the fault condition of the realization. This functional relationship is denoted as $Z(X,F)$ where $F \epsilon \mathfrak{F}_M$ and M is the module which realizes $Z(X)$ in the absence of faults.

Definition 3.3: $T = \{t_1, t_2, \ldots, t_s\}$, a set of input vectors, is a multiple pin fault test set for module M if, for each $F_j \epsilon \mathfrak{F}_M$, there exists at least one $t_i \epsilon T$ such that

$$Z(t_i, F_j) \neq Z(t_i, \phi).$$

The multiple pin fault set contains $3^{\ell+1} - 1$ elements, hence we strive to avoid the consideration of faults from this set on an individual basis. The following definitions are useful in discussing various faults.

Definition 3.4: A fault from the multiple pin fault set is comprised of components. Each line which is assigned a non-normal value, i.e. stuck-at-1 or stuck-at-0, is a component of the fault. A multiple pin fault may

be thought of as a set of single faults existing simultaneously, with each single fault being a component.

Definition 3.5: A fault component is detected completely by some test vector t_i if

$$Z(t_i,F_j) \neq Z(t_i,\phi)$$

for all F_j which contain the fault component.

One way to avoid consideration of all the elements in $\mathfrak{F}_M$ is to consider all of the possible fault components, and generate a test set which completely detects each of these fault components. The following theorem forms the basis of such a strategy.

Theorem 3.1: $T = \{t_1,t_2,\ldots,t_s\}$ is a multiple pin fault test set for module M realizing function Z(X) if for every input variable x_i there exists a pair of test vectors t_j, $t_k \epsilon T$ such that t_j and t_k differ only in coordinate i and $Z(t_j,\phi) \neq Z(t_k,\phi)$.

Proof: Any fault F which contains the output Z as a component will be completely tested by the pair t_j,t_k. If a component of some fault F is input x_i stuck at either 1 or 0, the internal logic network of module M will receive the same input vector for both t_j and t_k, since t_j and t_k differ only in coordinate i. Hence $Z(t_j,F) = Z(t_k,F)$. Since $Z(t_j,\phi) \neq Z(t_k,\phi)$ then either $Z(t_j,F) \neq Z(t_j,\phi)$ or $Z(t_k,F) \neq Z(t_k,\phi)$ hence a fault component on line x_i is completely detected by either t_j or t_k. Q.E.D.

Corollary: At most 2ℓ and at least $\ell+1$ tests are required to satisfy Theorem 3.1, where ℓ is the number of input pins.

Proof: Obviously no more than 2ℓ tests are required; however fewer than 2ℓ may be sufficient if it is possible to select test vectors so that some vectors are common to 2 or more of the test pairs for some of the variables. If maximal sharing is possible, then after the first variable is tested with a pair of tests, each of the remaining ℓ-1 variables may be tested by using one of the previously selected test vectors plus one new test vector thus requiring a total of $\ell+1$ tests. Q.E.D.

Functions which can be tested in accordance with Theorem 3.1 and meet the lower bound given in the corollary are numerous. Obvious examples are functions which are represented as a single minterm and functions containing a minterm which is also a prime implicant. The test sets for these functions contain the vector which is a prime minterm plus the ℓ input vectors which are obtained from the prime minterm by complementing one variable.

Theorem 3.2: Any module which implements a nonvacuous function of ℓ variables possesses a test set satisfying the hypothesis of Theorem 3.1.

Proof: If no such test set exists for function Z(X) which depends nonvacuously on $x_1, x_2, \ldots, x_\ell$, then there must be some variable, x_i, for which

$$Z(\alpha_1, \alpha_2, \ldots, \alpha_{i-1}, 0, \ldots, \alpha_\ell) = Z(\alpha_1, \alpha_2, \ldots, 1, \ldots, \alpha_\ell)$$

for all possible assignments of $\alpha_1,\alpha_2,\ldots,\alpha_{i-1},\alpha_{i+1},\ldots,\alpha_\ell$. If this is so then Z does not depend upon x_i. The theorem follows by contradiction.

Q.E.D.

Theorems 3.1 and 3.2 characterize a multiple pin fault test set and establish that such a test set always exists. These concepts can be formulated into a test generation procedure. A primary goal in the development of this procedure is to minimize the computational complexity of the procedure. In general, a function will possess several test sets of the type defined by Theorem 3.1. The generation of a test set of minimum size, i.e. one with the fewest test vectors, requires an examination of all of the test sets which exist for a particular function. Minimization of test generation complexity and minimization of test set size are therefore conflicting goals. The number of module pins, $\ell+1$, is a parameter which is primarily determined by integrated circuit technology. Manufacturers strive to reduce the module pin count to the minimum in order to increase manufacturing yields. Thus the number of inputs is limited to a relatively small number. As a result, the reduction of $|T|$ from 2ℓ to something in the range $\ell+1 \leq |T| < 2\ell$ is of questionable value and the additional complexity in generating a minimum or near minimum test set may not be justifiable in many practical situations. Minimum test sets will be discussed later in this chapter; however the main algorithm of this chapter has been developed with the goal of minimum test generation complexity without regard for reducing the size of the set below 2ℓ.

Before developing the details of a test generation algorithm, it is necessary to make some assumptions about how modules are specified,

since this information will be the input to the algorithm. Furthermore, preprocessing of this input could become the dominating factor in the running time of the algorithm. Since the trend toward more and more automation of the design of digital systems will undoubtedly continue, it is reasonable to further assume that module description will be available from the automated design process. Many recent design automation systems generate an irredundant cover of prime implicants at some point. We therefore assume that the input to the diagnostic test generation phase will proceed from this type of module specification.

Throughout the thesis it will be convenient to use the "cube representation" to discuss combinational functions. The universe of ℓ Boolean variables forms an ℓ-dimensional space in which each coordinate can take on the value 0 or 1. Each lattice point or vertex in this ℓ-dimensional cube represents one of the 2^{ℓ} possible input vectors. A particular function may be represented in this space by specifying the set of vertices (input patterns) for which the function takes on the value 1. Such vertices are referred to as true vertices, and the set of all true vertices of a function is called the "ON set." Similarly, vertices which are not true vertices are called false vertices, and make up the "OFF set."

Every true vertex corresponds to a minterm of the function. A collection of true vertices which forms a subcube corresponds to an implicant of the function. Implicants may be represented very conveniently by specifying an ℓ-tuple vector, each element of which is "0," "1," or "u." A "1" or a "0" in position i means that input variable x_i must be true or

false respectively in order for an input vector to be a vertex of the implicant; a "u" in position i indicates input variable x_i may be 0 or 1, i.e. the value appearing on input i is irrelevant to whether the input vector is a vertex of the implicant.

Definition 3.6a: A specified variable of a cube is a variable which is constrained to either a 0 or a 1 within the cube, i.e. all vertices of the cube have the same value for that variable.

Definition 3.6b: An unspecified variable of a cube is a variable which is not specified, i.e. the value of that variable is irrelevant to whether or not a vertex is a vertex of the cube. If a variable is unspecified for a cube, then half of the vertices of the cube will have a 0 in that coordinate, and half will have a 1.

Example 3.1: Implicant (for a 4 variable function)	cube notation
$\bar{x}_1 \bar{x}_2 x_3 \bar{x}_4$	0010
$x_1 \bar{x}_3$	1u0u

Using this notation it is quite convenient to specify a Boolean function as the set of cubes which imply the function.

Example 3.2:

$$Z(x_1,x_2,x_3,x_4) = x_1x_3 \vee x_2x_3x_4 \vee \bar{x}_1\bar{x}_2\bar{x}_3 = \{1u1u,\ u111,\ 000u\}$$

It is assumed that function specification in the form of a set of prime cubes is available as input to the test generation algorithm.

The following definitions will be useful in the discussion of test generation:

Definition 3.7: A vertex v_i is adjacent to a vertex v_j in variable x_i if v_i and v_j are identical in all positions except position i.

Definition 3.8: An alternating adjacency about variable x_i is a pair of vertices of the ℓ-space which are adjacent in variable x_i and have the additional property that one of the pair is a true vertex while the other is a false vertex.

Definition 3.9: An r-cube is an r-dimensional subcube of the ℓ-cube, and can be specified by an ℓ-element vector containing r unspecified and ℓ-r specified variables.

In order to generate a test set, it is necessary to find an alternating adjacency for each input variable. In addition, it is desired that the minimal amount of computation be required to accomplish this. The following theorem, though quite obvious, is important because it indicates the usefulness of a given prime implicant in terms of test generation.

Theorem 3.3: If P_i is a prime implicant of Z(X), and P_i is an r-cube, then there exists at least one alternating adjacency about each of the ℓ-r specified variables of P_i.

Proof: Assume there exists a specified variable, x_j, of P_i such that there is no alternating adjacency about x_j. Then since all vertices of the r-cube

obtained by complementing x_j in P_i are true, the (r+1)-cube consisting of P_i and the r-cube obtained by complementing x_j in P_i implies Z. Thus P_i is not prime. The theorem follows by contradiction. Q.E.D.

3.3. The Test Generation Process

The significance of Theorem 3.3 to test generation is that given a prime implicant of the function under consideration it is known that a test can be generated for each of the specified (0 or 1) variables. Thus the strategy will be to select a set of prime implicants which is sufficient to generate alternating adjacencies for all of the input variables. This selection is critical to reduction of computation, and requires the formation of a "cover."

Definition 3.10: A covering set of prime implicants is a set of prime implicants which has the property that for each variable, at least one member of the set has that variable specified (either 0 or 1).

The existence of a covering set is assured since it is assumed that the function depends nonvacuously on all variables. A covering set of minimum or near minimum size is desirable because this will reduce the number of cubes which must be considered. Fortunately this covering problem is much easier computationally than an arbitrary covering problem. Since a minimum cover is not essential, an efficient heuristic algorithm can be used.

Once a covering set has been generated, alternating adjacencies for each of the inputs can be generated using only prime implicants which are members of the covering set.

Consider a module which implements $Z(X) = \Sigma(P_1, P_2, \ldots, P_q)$ where P_i is a prime implicant. Let C be a covering set for Z, and consider $P_k \epsilon C$. Permuting the variables for clarity, it is possible to write P_k as follows:

$$P_k = u\ u \ldots u\ \alpha_{r+1} \ldots \alpha_\ell$$

where

$$\alpha_j \epsilon (0,1)$$

P_i is an r-cube. An alternating adjacency for variable x_i $(i > r)$, representing a test pair which completely detects the fault components x_i stuck-at-0 and x_i stuck-at-1, can be generated as follows. The r-cube which is adjacent to P_k in variable x_i is given by:

$$\tilde{P}_k^i = u\ u \ldots u\ \alpha_{r+1} \ldots \alpha_{i-1}\ \bar{\alpha}_i\ \alpha_{i+1} \ldots \alpha_\ell .$$

Theorem 3.4 guarantees that at least one vertex in $\tilde{P}_k^i$ is a false vertex (i.e. not a vertex of any of the other prime implicants of Z(X)). This set of false vertices in $\tilde{P}_k^i$ can be determined by performing the following operation:

$$\tilde{P}_k^i \wedge \bar{Z}.$$

The tests for x_i are generated as follows:

let

$$v = \beta_1 \beta_2 \ldots \beta_r\ \alpha_{r+1} \ldots \bar{\alpha}_i \ldots \alpha_\ell$$

where

$$v \epsilon p_k^{\tilde{i}} \wedge \bar{Z}$$

$$\beta_j \epsilon (0,1)$$

then the test pair for x_i is given by

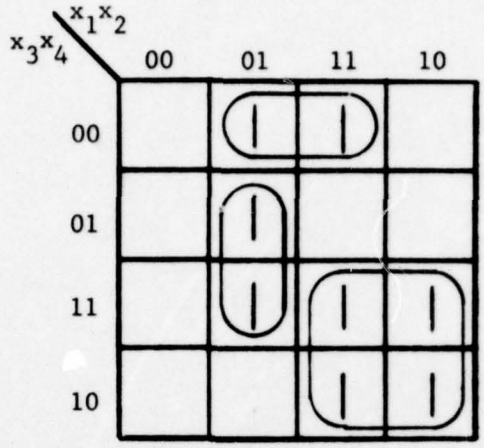
$$(\beta_1\beta_2\ldots\beta_r\ \alpha_{r+1}\ldots\bar{\alpha}_i\ldots\alpha_\ell, \beta_1\beta_2\ldots\beta_r\ \alpha_{r+1}\ldots\alpha_i\ldots\alpha_n).$$

In the above expressions, the α's are fixed so that the test vectors are in p_k and $p_k^{\tilde{i}}$, while the β's are selected to force the vertex in $p_k^{\tilde{i}}$ to be an element of $\bar{Z}$. The constraints which the β's are selected to meet may not require specification of all of the r unspecified variables in $p_k^{\tilde{i}}$. If this is the case, then there are several choices for test pairs. The following simple example demonstrates these ideas graphically using Karnaugh maps.

As the above discussion suggests the test generation process requires two main functions: the generation of a covering set, and the determination of one false vertex within a subcube of the ℓ-cube of the function. It is convenient to examine these two functions as separate problems, and then incorporate them into the main algorithm.

In the remainder of this chapter, specific algorithms will be discussed. The specification of these algorithms, which appear in this section and in the Appendix, is given in "Algol-like" notation. Key words of the specification language, e.g. do, if, and begin, are underlined.

Example 3.2: Test generation process.



function: $Z(x_1,x_2,x_3,x_4) = \Sigma\{P_1,P_2,P_3\}$

$= \Sigma\{u100,01u1,1u1u\}$

covering set: $C = \{P_1 = u100,\ P_2 = 01u1\}$

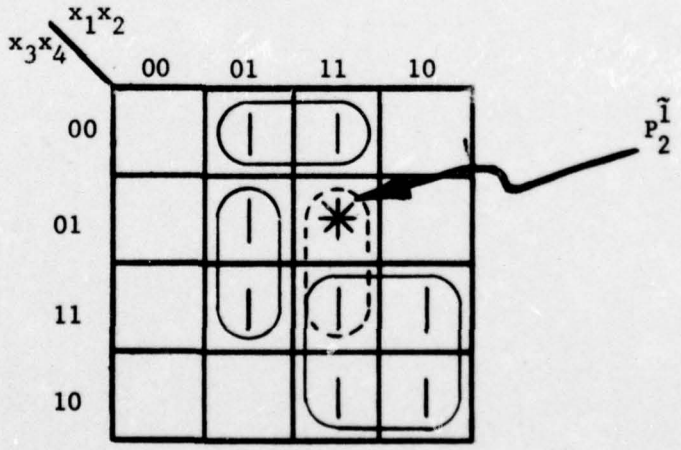
generate tests for x_1:

use $P_2 = 01u1$

$\tilde{P}_2^1 = 11u1$

$\tilde{P}_2^1 \wedge \overline{Z} = 1101$ (indicated by the asterisk in the map below)

tests for x_1 = ($1101 \in \overline{Z}$, $0101 \in Z$)



The symbol "←" denotes the assignment operator, while the symbol "=" denotes the relation "equal to" which appears in predicates of conditional statements. All algorithms are expressed in structured form. In these algorithms, arrays are used to represent the various multi-element quantities, and a particular element of an array is denoted by placing its subscripts in parenthesis. The symbol "*" used as a subscript denotes a dimension which is allowed to vary over its entire range. As an example, the set of prime implicants $\{P_i\}$ will be stored in the rows of two-dimensional array P. The j-th element of prime implicant P_i may be accessed in this array by the specification P(i,j), and the entire prime implicant P_i is accessed by the specification P(i,*).

3.3.1. Covering Set Generation

The covering problem which is required for the test generation algorithm may be stated as follows. Given a list of prime implicants, find a subset of them such that each variable is specified in at least one member of the set. An irredundant cover is desired, i.e. one in which each member in the set covers a variable not covered by any other member of the set. An irredundant cover guarantees that each prime implicant which is examined will yield at least one alternating adjacency. As will become apparent, small prime implicants, i.e. those with many specified variables, are desirable because a large number of tests can be generated from them, and because their adjacent cubes will intersect fewer other prime implicants. The size property of a prime implicant can be used to assign

a "cost" to it; the larger the prime implicant, the higher the cost of using it to form the covering set. We then use the cost measure to generate a "reduced cost" irredundant cover, which tends to favor usage of the smaller prime implicants. Since small prime implicants contain more specified variables, this cost objective is compatible with the objective of forming a cover with minimal or near minimal number of members. The basis for the cover generating strategy is a heuristic developed by Batni, et al. [19].

The following conventions will be employed throughout the following discussion. For vectors v and w of the same dimension and relation $R \in \{=, <, \leq, >, \geq\}$, vRw if and only if v(i) R w(i) for all i, and $v \not R w$ if and only if v(i) R w(i) for some i. The transpose of vector v is v'.

Consider an ℓ-variable function Z(x) which is represented by a set of q prime implicants. Let P be a $q \times \ell$ matrix which is used to store this set of prime implicants. For the covering problem considered here, variable i is covered by prime implicant j if the i-th component of j is specified, i.e. either a 0 or a 1, and is not covered if the i-th component is unspecified, i.e. "u." Let the symbol "s" denote a 0 or a 1 specification. Thus $P(i,j) \in \{s,u\}$. If P(i,j) = s then prime implicant P(i,*) covers variable j, and if P(i,j) = u then P(i,*) does not cover variable j.

The covering algorithm is more easily understood if the generation of an irredundant cover without regard to cost is first considered.

A weight is associated with each column of P. Vector W is an ℓ-element vector which stores this weight for each column. The weight for column j is defined to be the minimum row index i such that P(i,j) = s. That is, the weight is the index of the first prime implicant in the list which covers that variable. The cover is formed in q-dimensional vector CS where $CS(i) \in \{0,1\}$. CS(i) = 1 if prime implicant i is a member of the cover, and 0 otherwise.

Algorithm 3.1: Irredundant cover.

```
Begin  Basic Cover
     CS ← (00...0)          /* initialize cover to empty */
     do for j = 1 to ℓ      /* form initial weight vector */
          i ← 1
          do while P(i,j) = u
               i ← i+1
          end
          W(j) ← i
     end
     do until W = 0         /* form cover */
          k ← max[W(1),W(2),...,W(ℓ)]
          CS(k) ← 1         /* select an element for cover */
          do  i = 1 to ℓ    /* update weight vector */
               if P(k,i) = s, then  W(i) ← 0
          end
     end
end   Basic Cover
```

Algorithm 3.1 generates a cover which is irredundant because it selects as the next element of the cover the row which has the highest value in W. This row will be the only row which covers the variable corresponding to the index of the highest weight in W. Because of the manner in which the weight is defined, all rows which precede this row in P have a u in that variable position. After the row is chosen, W is updated. The next row selected will precede all previous rows selected. This guarantees that for each row selected, there is a variable which is covered by only that row. Thus each row in the final cover is essential to the cover.

It is seen that Algorithm 3.1 favors selection of rows with low indices. The cost constraint can therefore be introduced by ordering the rows in matrix P by ascending cost. This ordering may be accomplished by introducing a permutation of (1,2,...,q) which is stored in q-dimensional vector R. Let C be a q-dimensional vector with C(i) containing the number of unspecified variables of prime implicant i. Then R is formed such that $C(R(i)) \leq C(R(j))$ if $i < j$. R is introduced into Algorithm 3.1 so that rows are considered according to their position in vector R.

It is quite possible that the design algorithms which produced the function under consideration generated the prime implicant representation with a known ordering with respect to size. If this is the case, then determining the cost permutation R is a simple matter. If such is not the case, then a sort must be performed in order to determine R. Algorithm 3.2 encorporates these ideas to form a reduced cost irredundant covering set.

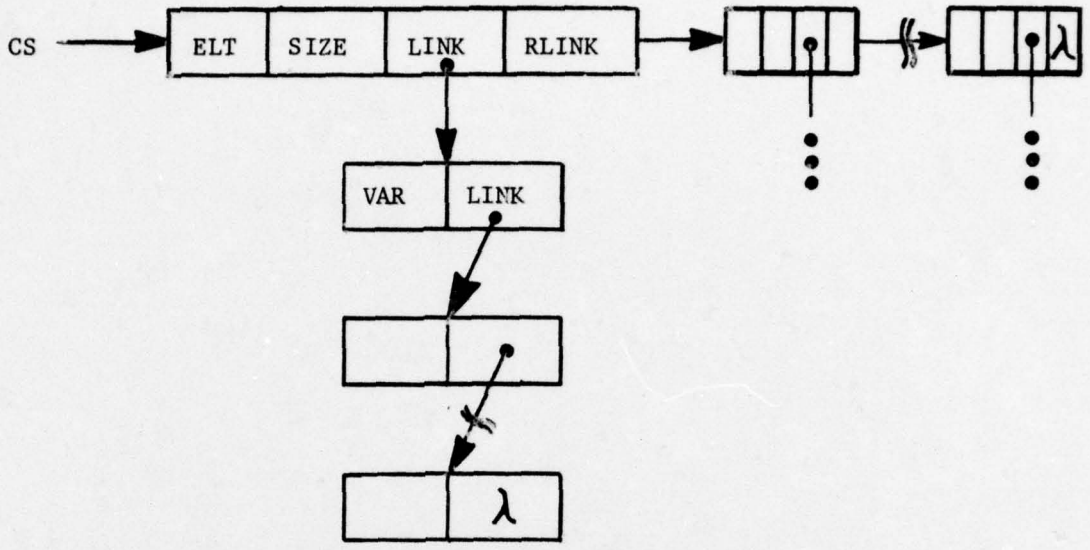
Algorithm 3.2: Reduced Cost Covering Set

```
Begin   COVER
     C ← [0,0,...,0]                        /* compute cost vector */
     do  i = 1 to q
          do  j = 1 to ℓ
               if  P(i,j) = s then          /* cost for P(i,*) = # of s's */
               C(i) ← C(i)+1
          end
     end
     do  i = 1 to q
         R(i) ← i                  /* initialize R to the natural order */
     end
     SORT (1,q+1)                  /* form the permutation R so that
                                      C(R(i)) ≤ C(R(j)) if i < j */
     do for  j = 1 to ℓ                  /* form weight vector, considering */
          i = 1                          /* rows in the order specified */
          do while  P(R(i),j) = u        /* in R. */
               i ← i+1
          end
          W(j) ← i
     end
     CS ← [0,0,...,0]
     do until  W = [0,0,...,0]           /* form cost reduced cover. */
          k ← max[W(1),W(2),...,W(ℓ)]
          CS(R(k)) ← 1
          do  i = 1 to ℓ
               if   P(R(k),i) = s, then  W(i) = 0
          end
     end
end  COVER
```

Algorithm 3.2 calls subroutine SORT which produces the permutation of indices corresponding to ascending cost. This subroutine is given in the Appendix.

This algorithm is incorporated into the main test generation algorithm for the purpose of determining which prime implicants can be used to generate tests for the various input variables. If Algorithm 3.2 is used as given above then later it is necessary to repeat some of the work already done in order to determine which variables to test with a given member of the covering set. This extra computation may be avoided by an extension to Algorithm 3.2. This modification results in a covering set data structure tailored to the needs of the main algorithm.

When an element of the cover is chosen and the weight vector is updated to reflect this choice, those elements of the weight vector which are nonzero, but which will be set to zero because this current choice covers those variables, correspond to the variables which can be tested by this prime implicant but could not be tested by any previously generated member of the covering set. By storing this information at the time each element of the covering set is chosen, we avoid having to keep track of which variables remain to be tested and which variables can be tested with the next prime implicant. It is also required to know the size (i.e. number of unspecified variables) of each member of the covering set. This enumeration is also performed as a byproduct of forming a covering set. Figure 3.2 illustrates a data structure designed to efficiently store these items of information as they are generated.



ELT: index into array P (set of prime implicants) of an element of the covering set.

SIZE: number of unspecified variables in this member of the covering set.

LINK: pointer to list of indices of variables covered by this element of the covering set which are covered by no previous (in the list) element of the covering set.

RLINK: pointer to list entry for the next element of the covering set.

VAR: index of a variable ($1 \leq VAR \leq \ell$) which is covered by element of the covering set indicated by the parent list entry.

LINK: pointer to next entry in list of covered variables.

Figure 3.2. Data structure for storing covering set.

The details of the modification to Algorithm 3.2 which builds the data structure are given in the Appendix. Later in this chapter when the test generation algorithm is discussed, it will be assumed that COVER provides, as output, the structure of Figure 3.2.

3.3.2. Determination of a False Vertex

Test generation hinges on the determination of a false vertex which is adjacent to some prime implicant. This problem may be stated in a general way as follows. Given a cube of dimension m and a set of subcubes, determine a vertex in this m-cube which is contained in none of the subcubes. The m-dimensional cube is the cube adjacent to the prime implicant (of size m), and the set of subcubes under consideration is the set of all non-null intersections of the other prime implicants with this m-cube. This problem is interesting from a computational viewpoint in its own right. In this section we develop an algorithm called FALSE VERTEX which performs this function. The basic algorithm will be developed in its general context first and will then be adapted to take on the structure of a subprocedure which is called from the main test generation algorithm.

The problem of determining a false vertex is basically a combinatorial search of the vertices of the m-dimensional cube.

In many search problems, it is convenient to represent the elements of the search domain as nodes of a tree such that the visitation of the elements corresponds to a traversal of the tree. The search algorithm determines which edge is followed from the node currently being visited. The search tree corresponding to this problem has the m-dimensional

universe (uu...u) as its root, and each node is a cube which represents a partial solution. Each edge corresponds to the assignment of a 0 or 1 to one of the unspecified positions of the preceding node. The terminal nodes are cubes containing one vertex, that is cubes having every position specified. The search terminates when a node, not necessarily terminal, is found which is disjoint from every cube in the list of true cubes. The position of a particular cube in the tree is of course determined by the search strategy. In this section, two search strategies will be presented. The first search is an exhaustive one and the second is a search by bisection.

An exhaustive search may be systematically conducted using the familiar "backtrack" or "depth-first" approach. The initial partial solution is the universal, uu...u. A pointer into the cube list is maintained, and the "next" partial solution is generated from the previous one by specifying (to be either 0 or 1) one of the u's so that the cube currently pointed to is in conflict (the intersection is null) with the trial solution. The pointer is then advanced to the next cube which does not yet conflict. If none of the as yet unspecified variables in the solution may be specified to create a conflict with the current cube, we backtrack to the previous trial solution and try a different partial solution. Thus at every point in the execution of the algorithm the current partial solution is disjoint from every cube in the list whose index is less than the pointer. The algorithm terminates with a solution when the pointer has advanced past the last entry in the cube list.

Algorithm 3.3: False Vertex-exhaustive.

C(i,j) is a list of v cubes of dimension m.

F(i) is the trial solution.

S is a stuck. Each element of the stack is a pair of indices.

i,j => S is a "push" operation

i,j <= S is a "pop" operation

```
Begin  FALSE VERTEX
     F ←(uu...u)                    /* initial partial solution */
     i ← 1                          /* index into cube list */
     do while i ≤ v
          if  F∩C(i,*)¬ = ∅   then do
               j ← 1
               do until  (C(i,j)∈{0,1}) ∧ (F(j) = u)
                    j ← j+1
                    do while j > m          /* backtrack */
                         i,j <= S
                         F(j) ← u
                         j ← j+1
                    end
               end
               F(j) ← ¬ C(i,j)        /* set F(j) so that F∩C(i,*) = ∅ */
               S <= i,j               /* save for subsequent backtrack */
          end
          i ← i+1
     end
end   FALSE VERTEX
```

In Algorithm 3.3, each time a previously unspecified variable is specified, we advance to a lower node in the search tree. At the time this next successor node is chosen, it is not known whether the subtree beneath this node contains a solution. If it does the search will find one, however if it does not the search will examine every node in this subtree and then backtrack out of this subtree and enter a different subtree of the preceding node.

By structuring the search in a slightly different way, it is possible to determine whether or not a subtree contains any solutions without having to search the subtree. If this determination is made at each stage of the algorithm, and the search procedure only enters a subtree known to possess a solution, then the algorithm will proceed directly to a solution without ever having to backtrack. To facilitate this process it is desirable for the search to be organized so that the search tree is binary. The result is a search by bisection, and we must only test one of the two subtrees beneath each node for the existence of a solution. If the subtree tested does not contain a solution, then it is assured that the other subtree must contain a solution. This is guaranteed because when the current node was entered the subtree consisting of this node and its two subtrees was tested for a solution.

The search may be organized to result in a binary search tree by restricting the choice of which variable position in the trial solution is to be specified at each stage of the algorithm. One way to do this is to require that the variables will be specified in a left-to-right order. The resulting search tree will have the property that all nodes at level i

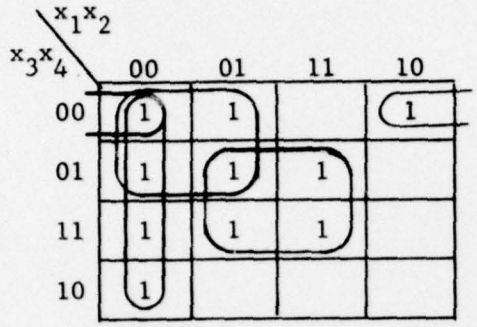
will have the first i variable positions specified and the remaining positions unspecified. Each node at level i will have two successors: that with the (i+1)st position specified to be a 0, and that with the (i+1)st position specified to be a 1. Each node of the tree represents some subcube of the original m-cube, and the successor nodes correspond to partitions of this cube.

In order to determine whether or not a successor subtree contains a solution, it is necessary to enumerate the true vertices contained in the subtree. If this number is less than the total number of vertices in the subtree, then there must be at least one false vertex in the subtree. This enumeration process is carried out by examining the cube list and counting the number of vertices which are in both the cube list and the partition of the m-cube corresponding to the subtree under consideration.

This enumeration process is nontrivial. First, the original cubes in the cube list must be preprocessed to obtain a set of cubes which cover the same vertices as the original list but which are mutually disjoint. This is done so that when vertices are enumerated, each true vertex in the subcube is counted only once.

A list of mutually disjoint cubes may be formed from an arbitrary list of cubes as follows. Starting with the first cube in the list and proceeding through the list, check for a nonnull intersection with each subsequent cube. When such an intersection is found, replace this cube with a list of cubes which covers all vertices not covered by the cube being considered. When all subsequent cubes have been checked, repeat the process for the next cube. The following example illustrates this process.

Example 3.4: Formation of disjoint cubes.



$C_1 \cap C_2$ = 00uu ∩ 0u0u
= 000u
≠ ∅

replace C_2 in the list with $C_2 - C_1 \cap C_2$ = 0u0u - 000u
= 010u

$C_1 \cap C_3$ = 0000

replace C_3 with $C_3 - C_1 \cap C_3$ = u000 - 0000 = 1000

$C_1 \cap C_4$ = ∅

new cube list is 00uu
010u
1000
ulul

Now compare C_2 to subsequent cubes:

$C_2 \cap C_3$ = ∅

$C_2 \cap C_4$ = 0101

replace C_4 with $C_4 - C_2 \cap C_4$ = ulul - 0101
= 1lul, x111

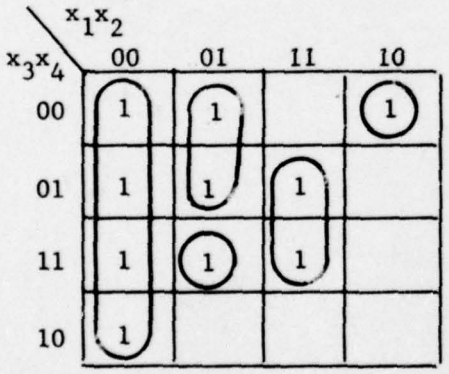
new cube list is: 00uu
010u
1000
11u1
u111

comparing C_3 to subsequent cubes:

$C_3 \cap C_4 = C_3 \cap C_5 = \phi$

Comparing C_4: $C_4 \cap C_5 = 1111$

$C_5 - C_4 \cap C_5 = 0111$



$C_1 = 00uu$
$C_2 = 010u$
$C_3 = 1000$
$C_4 = 11u1$
$C_5 = 0111$

list of mutually disjoint cubes

Since it may be necessary to create several new cubes to replace a cube, this procedure is most easily implemented using list processing techniques. The details of handling the various pointers results in a fairly lengthy specification of the algorithm. This algorithm is therefore presented in the Appendix.

The algorithm for determining a false vertex then proceeds using as input this list of disjoint cubes. The solution vector is initialized to uu...u. This vector is refined to yield a false vertex by specifying the variables from left to right. At each position a 0 is tried first and the subtree corresponding to this choice is tested to determine if a solution exists. If one does, then the next variable is specified. If not then a 1 is used instead of a 0. It is not necessary to test for a solution in the subtree corresponding to this choice, because it is already known from the test performed previously that either the 0 or the 1 subtree contains a solution. When all variables are specified by this procedure, the resulting vertex is guaranteed to be false. Every time a variable is specified, the cube list can be shortened, which enhances the efficiency of the algorithm substantially.

The algorithm which implements this strategy is a list processing algorithm. The specification is given in detail in the Appendix; an overview is presented here.

Algorithm 3.4: False Vertex - Bisection (abbreviated) (cubes contain m positions).

```
Preprocess cube list to get a list of disjoint cubes.
F ← (uu...u)                          /* F is the solution */
do for j = 1 to m
     count ← 0
     F(j) = 0                         /* try a zero in position j */
     scan the cube list
          if cube (j) = 0             /* cube ⊆ F */
             count ← count + cube size
          else if  cube (j) = 1       /* cube ∩ F = ∅ */
                   move cube to alternate list
          else                        /* cube (j) = u:  cube ∩ F ≠ ∅, but ⊄ F */
                   move cube to alternate list
                   count ← count + cube size
     end  scan
     if  count = 2^(m-j)  then do
         F(j) ← 1
         replace cube list with alternate list      /* this is all cubes in
                                                       the F(j) = 1 partition */
        null the alternate list
      end
      else  null alternate list       /* F(j) = 0 is o.k.  The cube list has
                                         been purged of all nonintersecting
                                         cubes */
end
```

In Algorithm 3.4, when the $x_i = 0$ choice is made and the cube list is scanned, any cubes which are properly contained in the $x_i = 1$ partition are removed from the cube list and placed in an alternate list. Any cube with a u in x_i intersects both the $x_i = 0$ and $x_i = 1$ partitions of the space. Such cubes are effectively split, and the $x_i = 0$ portion remains in the cube list while the $x_i = 1$ portion is placed in the alternate list. This process accomplishes two things. First, the cube list is purged of cubes which do not intersect, thus the cube list becomes shorter and shorter. Second, if the $x_i = 0$ choice is wrong, then the alternate list contains all cubes in the $x_i = 1$ partition and can be used to replace the cube list in subsequent variable selections.

3.3.3. Test Set Generation Algorithm

In this section we will discuss details of an algorithm which generates a multiple pin fault test set. As previously indicated, the first step is the formation of a covering set, CS. Then, for each prime implicant, P_i, in CS, alternating adjacencies for each specified variable which has not already been considered are generated. As previously discussed, the difficult part of this procedure is determination of a false vertex which is adjacent to the cube in the variable under consideration. This process is broken into two parts. First, we form $\tilde{P_i^j}$ and determine the set of prime implicants which have a nonnull intersection with $\tilde{P_i^j}$. Then given this set, we find a false vertex contained in $\tilde{P_i^j}$, that is a vertex which is in none of the intersections. In general, we will perform this process several times with a single P_i. The most

time consuming part of the process is scanning the set of prime implicants looking for intersections with $\tilde{P}_i^j$. It is possible to do this scan only once for each $P_i \epsilon CS$ regardless of how many $\tilde{P}_i^j$'s we generate from each P_i.

Definition 3.11: The distance between cubes C_1 and C_2 (of the same dimension), which is denoted by $d\{C_1,C_2\}$, is the number of variables in which C_1 and C_2 disagree. An unspecified variable does not disagree with either a 0 or a 1 specified variable. For example, $d\{001ulu,0110uu\} = 1$.

This distance metric indicates the "proximity" of two cubes within the ℓ-space, and is useful here because it indicates when nonnull intersections between cubes can occur. Three cases are of interest.

Case 1: $d\{C_1,C_2\} = 0$. This implies that C_1 and C_2 have some set of vertices in common, i.e. they overlap on a Karnaugh map.

Case 2: $d\{C_1,C_2\} = 1$. Here C_1 and C_2 are disjoint, however there is some variable which, if complemented in C_1 or in C_2, would result in a nonnull intersection between the two cubes.

Case 3: $d\{C_1,C_2\} \geq 2$. C_1 and C_2 are disjoint and there is no single variable change which could result in an intersection between C_1 and C_2.

Now consider $P_i \epsilon CS$ and the set of adjacent cubes $\{\tilde{P}_i^j\}$ which are used to generate alternating adjacencies (each cube in this set corresponds to some x_j for which tests are derived from P_i). By examining the distance between P_i and P_k, $\forall k \neq i$, we can determine if P_k could possibly intersect any of the $\{\tilde{P}_i^j\}$ because we know that $d\{P_i,\tilde{P}_i^j\} = 1$.

Case 1: $d\{P_i,P_k\} = 0$. P_i and P_k agree in every coordinate where both are specified. Since $d\{P_i,\tilde{P_i^j}\} = 1$ for all $\{\tilde{P_i^j}\}$, we know that $d\{\tilde{P_i^j},P_k\} = 1$ unless the j-th position of P_k contains a u. Thus there will be a null intersection between $\tilde{P_i^j}$ and P_j unless this case occurs.

Case 2: $d\{P_i,P_k\} = 1$. P_i and P_k disagree in exactly one coordinate where both are specified. In this case at most one of the $\{\tilde{P_i^j}\}$ will have a non-null intersection with P_k, and that will occur if j is the coordinate where P_i and P_k disagree.

Case 3: $d\{P_i,P_k\} \geq 2$. P_k will not intersect any of the $\{\tilde{P_i^j}\}$.

These observations about the significance of the distance relationship are used to speed up test generation for large problems, i.e. those with many prime implicants. Before test generation begins, an array of distances between each element of the covering set and all of the other prime implicants is formed. Then as each element of $\{\tilde{P_i^j}\}$ is formed, only those prime implicants which could possibly intersect elements of the sets $\{\tilde{P_i^j}\}$, i.e. those with $d = 0$ or $d = 1$, need be examined.

The size of each $\tilde{P_i^j}$ is of course equal to the size of the prime implicant, P_i, from which it was derived. The search for a false vertex takes place within the cube of $\tilde{P_i^j}$. Thus when a prime implicant which has a nonnull intersection with some $\tilde{P_i^j}$ is detected, we wish to form a cube equal to this intersection. Only the subspace corresponding to the $\tilde{P_i^j}$ is relevant. This subspace is determined by the unspecified positions of $\tilde{P_i^j}$. Therefore the test generation algorithm strips out the variable positions

corresponding to u's in $\tilde{P}_i^j$ when forming the cubes to be passed to FALSE VERTEX, and assimilates the values returned by FALSE VERTEX back into $\tilde{P}_i^j$ to form the false test of the test pair for x_j. From this test the true test is obtained by simply complementing variable x_j.

The details of TEST GEN are rather involved since much manipulation of the pointer fields in the data structure constructed by COVER is required. Therefore the specification of algorithm TEST GEN is again relegated to the Appendix. The algorithm is presented here in abbreviated form.

Algorithm 3.5: Test Set Generation (abbreviated)

```
begin  TEST GEN
     call  COVER
     call  DMATRIX              /* compute distance between prime implicants */
     do for (each prime implicant, P_i, in the covering set)
          do for (each variable, x_j, for which tests are to be generated
                  from prime implicant P_i)
               form list of cubes of intersection between P_k and P~_i^j for
                    k ≠ i
               call  FALSE VERTEX
               form test vectors for variable x_i
          end
     end
end  TEST GEN
```

3.4. Computational Complexity of Test Set Generation

In order to establish computational feasibility of the test set generation algorithm developed in Section 3.3 and compare it to other fault detection algorithms, it is worthwhile to analyze the "computational complexity" involved. Computational complexity of an algorithm is an algebraic expression which describes the relationship between some performance parameter, such as execution time or storage requirement, and the set of parameters which characterize the input to the algorithm. These relationships are inherent to the algorithm and therefore allow the quantification of performance without having to revert to an analysis of the programming details and instruction execution times of a particular machine. Since we are interested only in the dependence of performance on input parameters, we express complexity as "order" and use the symbol "O." Constants of proportionality and lower order terms, i.e. those which become negligible as the input parameters become large, are not expressed in order to simplify the complexity expression and show the dominating terms.

In this section the complexity of the test generation algorithm developed in Section 3.3 will be analyzed and compared to the complexity of test generation algorithms based on other fault models.

The performance of Algorithm 3.5 is highly dependent upon the specific function to be tested. While worst-case analysis yields unrealistically pessimistic results, it is not possible to compute expected performance since there is no basis for assuming particular probability distribution functions for the parameters describing the input. In the

discussion which follows, a worst-case analysis will be performed; however, some of the significantly problem-related coefficients will be expressed symbolically, and we will discuss how the nature of the input affects the values of these coefficients.

3.4.1. Execution Time of Test Set Generation

Algorithm 3.5 consists of two preprocessing components (COVER and DMATRIX) which are executed once, and the test generation block which calls FALSE VERTEX once for each pair of tests generated. FALSE VERTEX calls DISJOINT CUBES as an initialization step.

The following parameters are used to characterize the set of prime implicants which serves as input to TEST GEN.

ℓ = the number of input variables of the function to be tested.

q = the number of prime implicants used to represent the function to be tested.

c = the number of prime implicants in the reduced cost covering set generated by COVER. (Note that $c \leq \min(\ell, q)$.)

COVER first computes a cost vector by enumerating the unspecified variables in each of the prime implicants. This requires the examination of each of the ℓq variable positions. Formation of the cost-ordering permutation, R, requires sorting the cost vector, a task known to require $O(q \log_2 q)$ operations [20]. Computing the initial weight vector, W, requires at most ℓq examinations. Finding the maximum weight and assigning it to k can take at most $\ell+1$ steps. Updating the weight vector can require at most ℓ steps. The loop which consists of finding the maximum weight,

selecting the next member of the cover, and updating the weight vector is repeated once for each member in the covering set. Thus the execution time required by COVER is as follows.

$$T_{COVER} \propto \ell q + q \log q + \ell q + c(2\ell+1)$$
$$\propto 2\ell q + q \log q + 2\ell c + c$$
$$= O(\ell q + q \log q)$$ (Since $\log q$ and ℓ may be of the same order, both terms are retained.)

DMATRIX computes the distance, up to $d = 2$, between each of the c elements of the covering set and the q-1 other prime implicants. This requires no more than ℓqc steps.

FALSE VERTEX is passed a list of cubes in some subspace of the ℓ-cube. Assume that this list contains α cubes of size β. The cube list is first transformed into a list of disjoint cubes. Let this list contain α' cubes. Within the main loop of DISJOINT CUBES, the β positions of the cube currently being examined are tested either once if there is no intersection or twice if there is. This loop is executed at most $(\alpha'-1) + (\alpha'-2) + \cdots + 1 = \frac{\alpha'(\alpha'-1)}{2}$ times. The length of the list resulting from the application of DISJOINT CUBES may never exceed 2^{β}. Therefore DISJOINT CUBES could require no more than $O(2^{2\beta})$ steps to be executed. FALSE VERTEX then determines a false vertex in time bounded by the series $(\beta-1)(2^{\beta}-1) + (\beta-2)(2^{\beta-1}-1) + \cdots + 2^2$ which is clearly less than $O(2^{2\beta})$. Thus execution time of FALSE VERTEX is dominated by the processing required to make the cube list mutually disjoint. This analysis is extremely pessimistic since it is quite likely that α and α' will be $\ll 2^{\beta}$. β is at most $\ell-1$.

TEST GEN is comprised of two sequentially executed blocks, COVER and DMATRIX, and the main loop. This loop scans the data structure which has $c+\ell$ elements. For each of the ℓ elements which comprise the sublists, the list of prime implicants is scanned for nonnull intersection with $\tilde{P}_i^j$. This requires $0((q-1)\ell)$ steps. Then FALSE VERTEX is called. Thus time complexity of the main loop of TEST GEN is given by the following expression:

$$0(\ell[(q-1)\ell + \text{FALSE VERTEX} + \ell]) =$$
$$0(\ell^2 q) + 0(\ell 2^{2\beta}).$$

This results in an overall time complexity for TEST GEN of the following:

$$0(\text{TEST GEN}) = 0(q \log q) + 0(\ell q c) + 0(\ell^2 q) + 0(\ell 2^{2\ell}).$$

3.4.2. Storage Requirements for Test Set Generation

During the storage analysis we shall use the term "cell" to mean the number of bits required to represent the quantity in question plus any link information required. Since all cells contain $0(\ell)$ or fewer bits, we shall assume $0(\ell)$ bits are used for all cells.

During execution of the main loop of TEST GEN, there is an amount of static storage plus some time-varying storage. The static requirement consists of q cells for the array of prime implicants plus $c \cdot q$ cells for the distance matrix, plus $c+\ell$ cells for the list structure containing the covering set information. As TEST GEN is executed, a list of intersections of $\tilde{P}_i^j$ with P_k, $k \neq i$, is constructed and the list is made disjoint. This list can reach a maximum size of 2^β cells. Thus storage required is

as follows:

$$O[\ell(q + c \cdot q + c + \ell + 2^{\beta}] \approx O[\ell cq + \ell^2 + 2^{\ell}].$$

3.4.3. Discussion of Complexity

The complexity of TEST GEN is dominated by FALSE VERTEX which requires exponential processing time and storage in the worst case. This worst case is based on the assumption that the cubes passed to FALSE VERTEX cover all but one of the vertices of the β-dimensional subcube, and that they do so in such a way that after the list has been made mutually disjoint, each cube covers only one vertex. This is a very unusual cases, since as DISJOINT CUBES scans the list of cubes it always retains the cube it is examining and reshapes subsequent cubes of the list. When a cube is reshaped, the number of cubes generated to replace it is equal to the number of occurrences of an unspecified variable in the cube to be reshaped corresponding to a specified variable in the cube to be retained. The number of such occurrence is some fraction of β. If there are no such occurrences, then the cube is replaced by a null list resulting in a decrease in the length of the cube list. A cube is reshaped only when there is an intersection with a cube earlier in the list. Therefore one would expect the list expansion factor to be related to β rather than 2^{β} in the typical case.

Since the covering set generation algorithm selects prime implicants of small size by the cost reduction mechanism, the subcube size, β, tends to be minimized. Thus the complexity expressions developed

in the previous sections are misleading in all but certain pathological cases.

Although certain combinational functions can be irredundantly represented with a large number of prime implicants, it has been observed [21] that the types of combinational functions which often crop up in actual design are "shallow" functions, i.e. those which can be represented with a small ($0(\ell)$) number of prime implicants. The use of programmed logic arrays is becoming more and more common. Such devices implement functions comprised of a relatively small number of prime implicants because of constraints imposed by the techniques used to fabricate such devices. The number of inputs, ℓ, is also limited by integrated circuit technology to a relatively small number (~ 10 - 40). These observations support the computational feasibility of test generation based on the pin fault model.

In contrast to the multiple pin fault approach, the best algorithms known for detecting single faults under the stuck-line fault model have a computing time that is exponential in the number of input lines and gates [22]. As a result, test set generation under these fault assumptions is practical only for small circuits.

3.5. Minimum Size Test Sets

The test generation process described in Section 3.3 generates test sets of size 2ℓ where ℓ is the number of input pins. These tests have the advantage that they are relatively easy to generate and are symmetric, i.e. the output sequence resulting from application of the

test set is made up of ℓ zeros and ℓ ones. The symmetry property is advantageous because it enables the circuit to be tested without the necessity to store an array of fault-free circuit responses to the test inputs. Future development of the pin fault approach will probably be directed at testing networks of modules in an analogous way to the testing of gates under the stuck-line model. In this case reduction of the size of the test set, even though small, may be of value.

The determination of minimum test sets has proven to be a very difficult problem, and major results in this area have not been developed. In this section, a summary of results and observations related to test set reduction will be presented.

3.5.1. Test Sets Satisfying Theorem 3.1

The corollary to Theorem 3.1 states that it may be possible for a combinational circuit with ℓ inputs to possess a multiple pin fault test set which satisfies Theorem 3.1 with as few as $\ell+1$ test vectors. In order to consider the question of minimum test sets within the context of Theorem 3.1 it is useful to use the following graphical representation for functions. An ℓ-input function is represented in the ℓ-dimensional lattice by true vertices as solid dots and false vertices as circles. Figure 3.3 illustrates the lattice representation for $Z = x_1x_4 \vee x_3x_4 \vee x_1x_2\bar{x}_3 \vee \bar{x}_1\bar{x}_2x_3$. As any edge is traversed, one of the vertex coordinates is complemented. This property of the lattice is used to associate the edges of the lattice with the input variables as follows. Each edge is

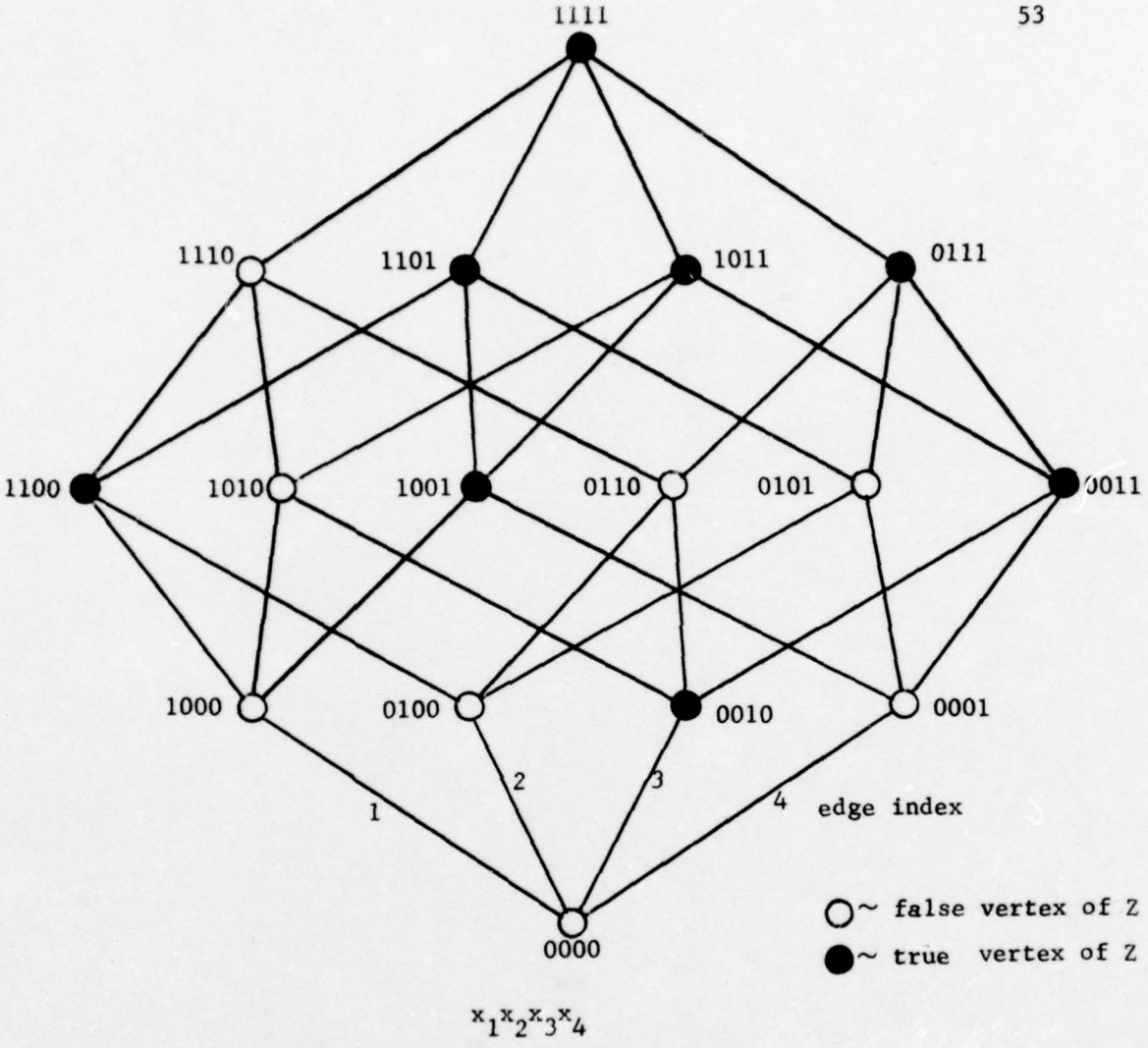


Figure 3.3. Lattice representation of $Z = x_1x_4 \vee x_3x_4 \vee x_1x_2\bar{x}_3 \vee \bar{x}_1\bar{x}_2x_3$.

assigned an "edge index" which is equal to the index of the input variable which is complemented as the edge is traversed.

If a false vertex is connected to a true vertex by an edge with index i then these two vertices correspond to an alternating adjacency about input variable x_i, and a set of such vertex pairs, one pair for each edge index i, $1 \leq i \leq \ell$, corresponds to a test set satisfying Theorem 3.1. This set of vertex pairs and joining edges represents some subgraph of the lattice representation. A minimum test set satisfying Theorem 3.1 corresponds to a subgraph of this type with the minimum number of vertices. Since every such subgraph must contain exactly ℓ edges, the subgraph with the fewest number of vertices is one with maximum connectivity. Cycles cannot occur in such a subgraph because in order to traverse a path beginning and ending with some vertex v_j, each variable position would have to be complemented an even number of times. Thus if an edge with index i occurs in the path then it must occur again so that input variable x_i would be tested more than once. Therefore a minimum test corresponds to a tree or set of trees. The subgraph of ℓ edges with the fewest number of vertices is a connected tree. Any connected tree with ℓ edges has $\ell+1$ vertices hence the lower bound given in the corollary to Theorem 3.1.

Definition 3.12: A minimal spanning tree (MST) for a function of ℓ variables is a subgraph of the lattice representation of the function with the following properties:

1. The subgraph is a connected tree.
2. For each i, $1 \leq i \leq \ell$, there is exactly one edge in the subgraph whose index is i.

3. For every pair of vertices v_i and v_j which are joined by an edge $Z(v_i) \oplus Z(v_j) = 1$.

(Note that this definition bears no relation to the usual minimal spanning tree concept of graph theory.)

Figure 3.4 shows two MST's for the function represented by Figure 3.3, and the corresponding test sets T_1 and T_2. (In Figure 3.4 only edges belonging to the MST's are shown.) Note that input x_i is tested by the two vertices joined by the edge whose index is i.

The following theorem is an immediate consequence of Definition 3.10 and Theorem 3.1.

Theorem 3.5: Module M realizing function $Z(x_1x_2\ldots x_\ell)$ possesses a test set satisfying the hypothesis of Theorem 3.1 with the minimum number of tests if and only if the lattice representation of Z contains an MST.

The lattice representation of functions together with the necessity clause of Theorem 3.5 allows many easily testable functions to be recognized. For example any function containing a prime implicant which is a minterm or a prime implicant which is a maxterm possesses a test set of size $\ell+1$ because in the lattice representation of such functions, an MST can be found surrounding the minterm or maxterm as illustrated in Figure 3.5. Another interesting class of functions is the class which can be realized by a network containing no fanout. The following theorem establishes this result, and its proof may be used to construct a test set for any such function.

T_1 = {1110,1100,1000,0100,1001}

T_2 = {1010,0010,0110,0111,0101}

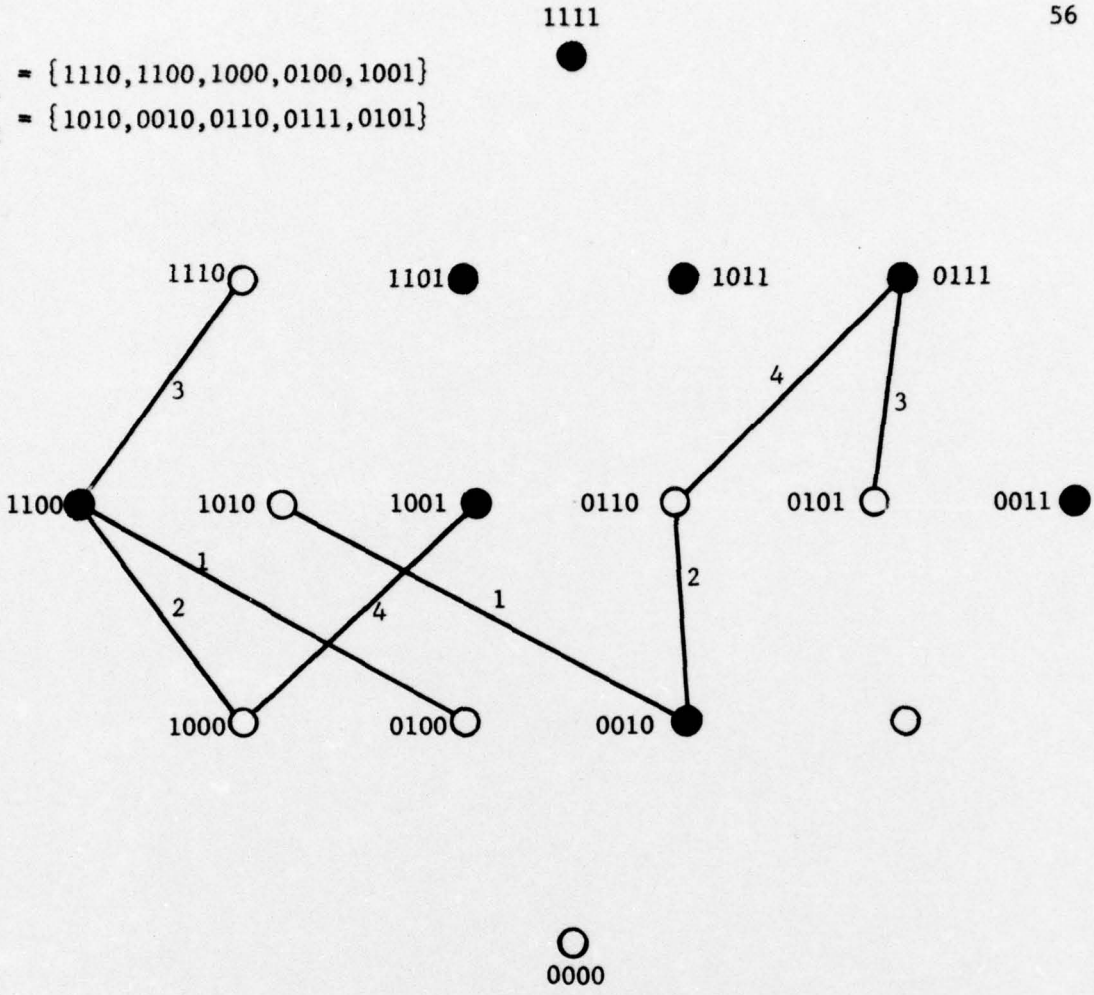
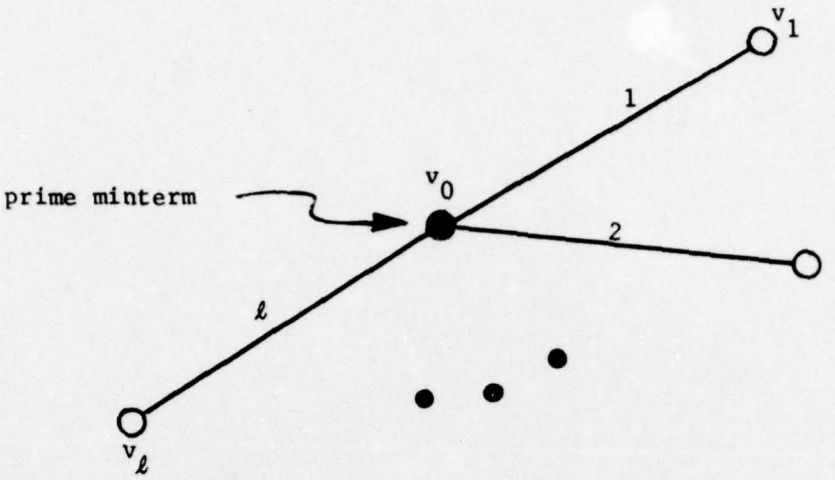
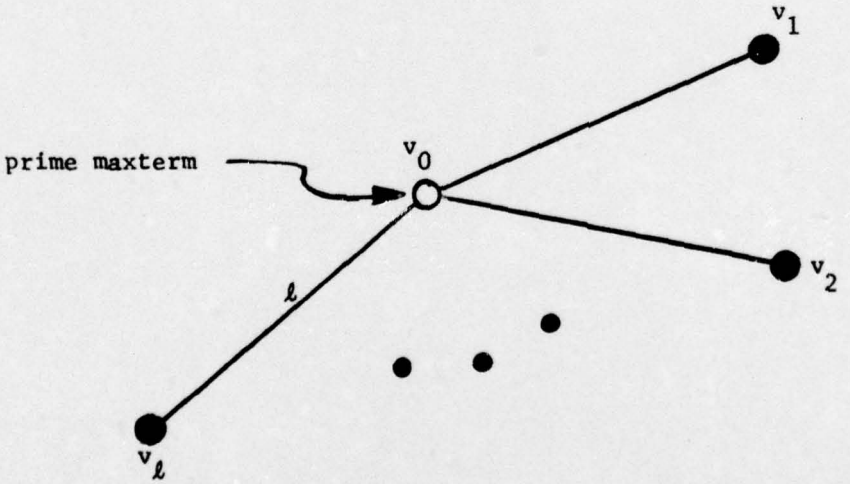


Figure 3.4. Two MST's and the corresponding test sets of the function represented in Figure 3.3.



(a)



(b)

Figure 3.5. MST's for functions with prime minterm (a) or maxterm (b).

Theorem 3.6: Any function which can be realized as a fanout-free (including inputs) network has a lattice representation containing an MST and therefore a test set of size $\ell+1$, where ℓ is the number of inputs.

Proof: (By induction on the number of gates in the realization.) Let T_k be a network of k AND, OR, NAND, and NOR gates which realizes function $Z(x_1x_2\ldots x_\ell)$ without fanout. Consider the degenerate case, T_1. Since T_1 possesses a single minterm (AND or NOR) or maxterm (OR or NAND) the function realized by T_1 possesses an MST of the type shown in Figure 3.5.

Assume that function Z is realized as network T_k, and that T_k possesses an MST. Form network T_{k+1} from network T_k by adding gate G as shown in Figure 3.6. (Note that the inputs to T_k may be permuted so that G drives line x_1 without loss of generality.) Consider the MST for T_k, and remove the edge whose index is 1. Figure 3.7 shows this edge. Let $v_i = \alpha_1\alpha_2\ldots\alpha_\ell$. Then $v_j = \bar{\alpha}_1\alpha_2\ldots\alpha_\ell$, $\alpha_i\epsilon\{0,1\}$. All vertices in the subtree headed by v_i have $x_1 = \alpha_1$. All inputs which are tested by vertices in the v_i subtree will still be tested if the inputs y_i to G are held constant at values such that $G(y_1y_2\ldots y_m) = \alpha_1$, and similarly for the v_j subtree with $G(y_1y_2\ldots y_m) = \bar{\alpha}_1$. Since G consists of one gate, it realizes a function with a single min- or maxterm, and has an MST of the type illustrated in Figure 3.5. The subtrees headed by v_i and v_j, and the MST for G may be combined into a single MST for T_{k+1}: The edge which formerly tested x_1 will test one of the inputs to G. The remaining inputs to G will be tested by new edges emanating from either v_i or v_j.

The pattern $\alpha_2\alpha_3\ldots\alpha_\ell$ when applied to $x_2x_3\ldots x_\ell$ sensitizes line x_1 to the output Z_{k+1}. That is, either $Z_{k+1}(x_1,\alpha_2,\alpha_3\ldots\alpha_\ell) = x_1$ or

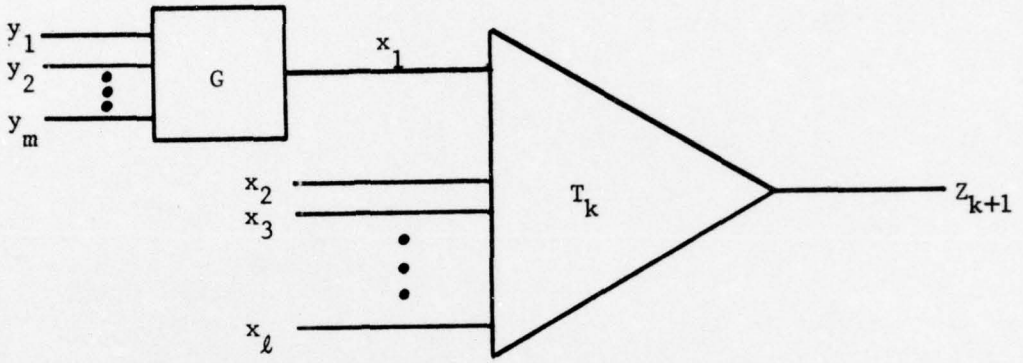


Figure 3.6. Network T_{k+1}.

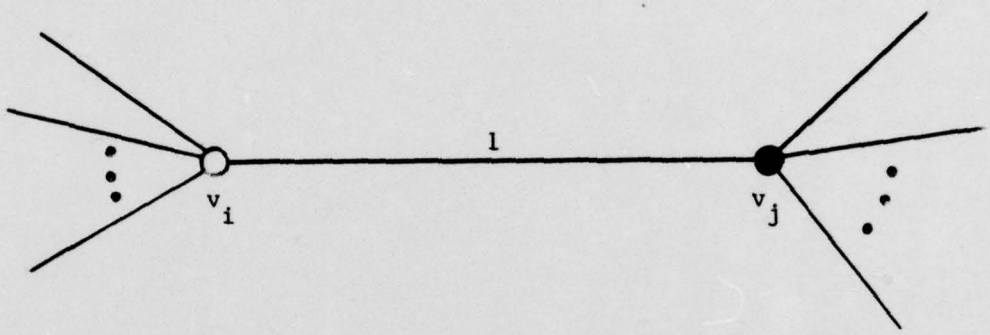


Figure 3.7. The MST edge corresponding to x_1.

$Z_{k+1}(x_1, a_2 a_3 \ldots a_\ell) = \bar{x}_1$. Therefore, the m inputs to gate G can be tested by applying the vertices of the MST for G to $y_1 \ldots y_m$ while holding $x_2 \ldots x_\ell$ constant at $\alpha_2 \alpha_3 \ldots \alpha_\ell$. The inputs $x_2 \ldots x_\ell$ are tested by holding $y_1 \ldots y_m$ constant at values resulting in $x_1 = \alpha_1$ to test those inputs in the subtree formerly headed by v_i and by holding $y_1 \ldots y_m$ constant at values resulting in $x_1 = \bar{\alpha}_1$ to test those inputs in the subtree formerly headed by v_j. The resulting tree has the structure shown in Figures 3.8a or 3.8b, and is an MST for network T_{k+1}. The theorem is thus established for a fanout free network consisting of any number of gates. Q.E.D.

Generation of an MST and therefore a multiple pin fault test set by the constructive method used in the above proof is very simple. The procedure begins at the output gate and proceeds by successive levels considering each gate until all inputs are reached. The MST for the output gate, Figures 3.5a or 3.5b, is drawn first. Then as each gate is considered, the MST for that gate type is added to the graph by replacing the edge which tests the gate's output by one of the edges of the MST for that gate. The tree structure is built up in this way so that when all input pins have been reached, only edges corresponding to tests for inputs will remain. Whenever a gate which is fed by primary inputs is encountered, the edges added are labeled with the input which the endpoints of that edge test. The result is an MST with labeled edges. The test vertices are then assigned (in a unique way) by considering each gate which receives primary inputs and writing down the test input required on those primary inputs for that gate type. These tests are the u- and e-tests of Hayes [23] for that

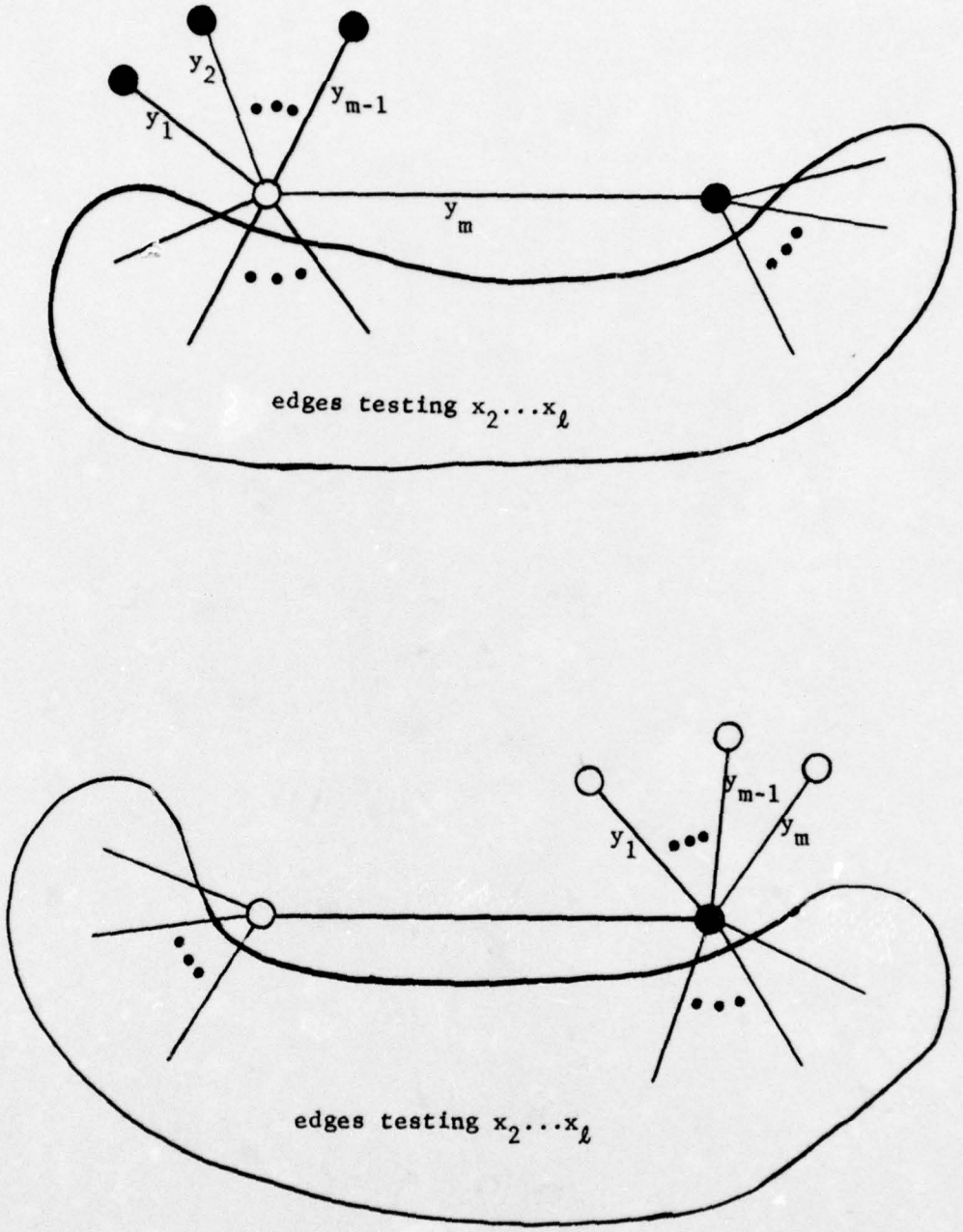


Figure 3.8. MST for T_{k+1}.

gate type as given in Table 3.1. When this is done for all inputs, all variable positions for each vertex in the MST will be assigned, and these vertices are guaranteed to constitute a multiple pin fault test set.

Table 3.1. MST Vertices for Gates

gate type	v_0	v_1	v_2	...	v_n
AND,NAND	11...1	01...1	101...1	...	11...10
OR,NOR	00...0	10...0	010...0	...	00...1

The following example illustrates construction of an MST and the corresponding test set by the above method. Figure 3.9 shows a 4-level network and Figure 3.10 illustrates construction of an MST. The final step of assigning values to the vertices is achieved by making the correspondence between each nonterminal vertex and a gate in the network. Then for each of these vertices, the values for the inputs tested by adjacent vertices can be assigned based on the type of gate. For example the vertex corresponding to gate NOR_6 has adjacent vertices which test its inputs, x_6, x_7, and x_8. Thus, since this gate is a NOR, the vertex labeled NOR_6 must have 000 in positions x_6, x_7, x_8, and the vertices connected by x_6, x_7, and x_8 must have 100, 010, 001 respectively in positions x_6 x_7 x_8.

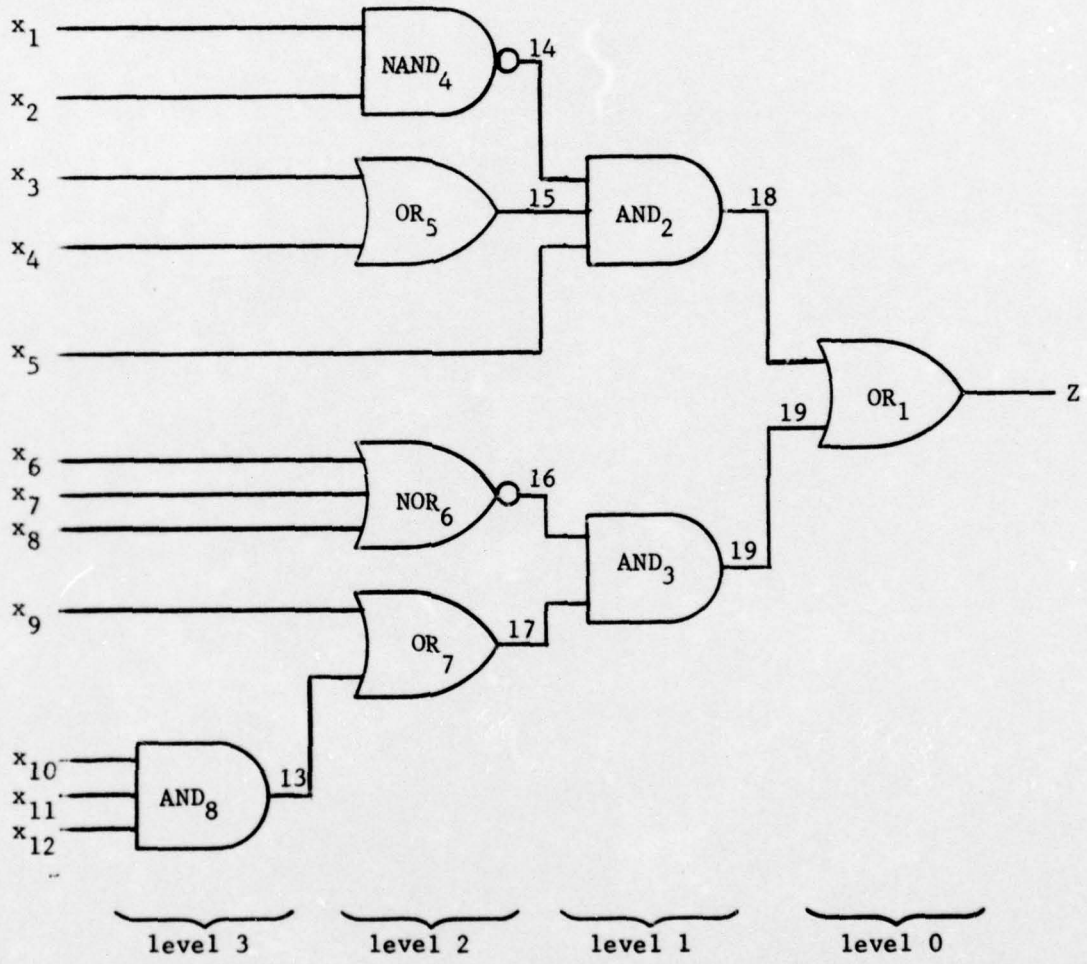
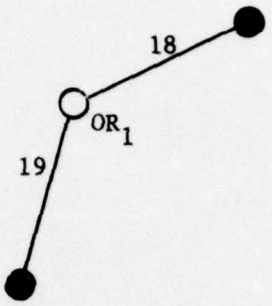
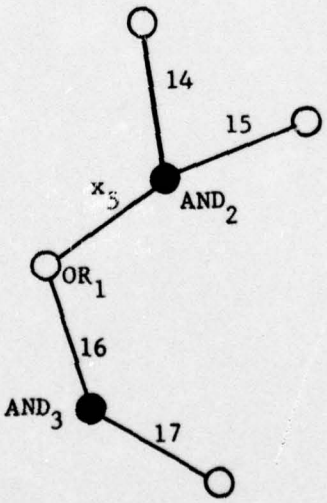


Figure 3.9. A 4-level fanout-free network.

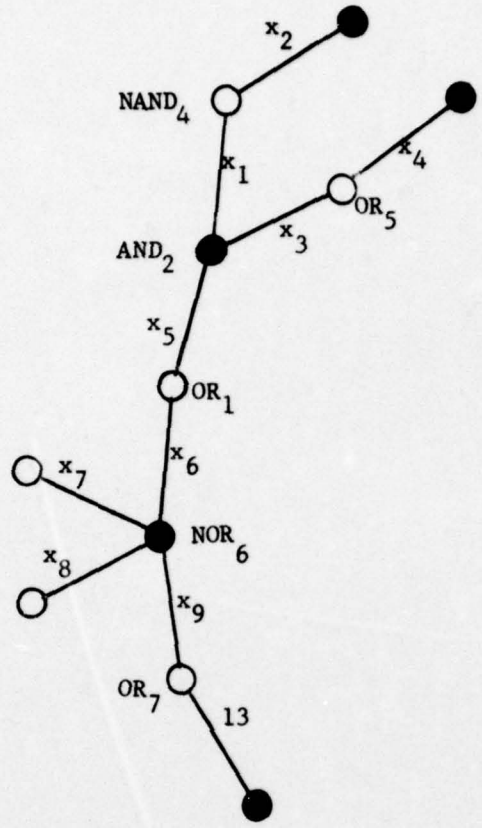


(a) level 0

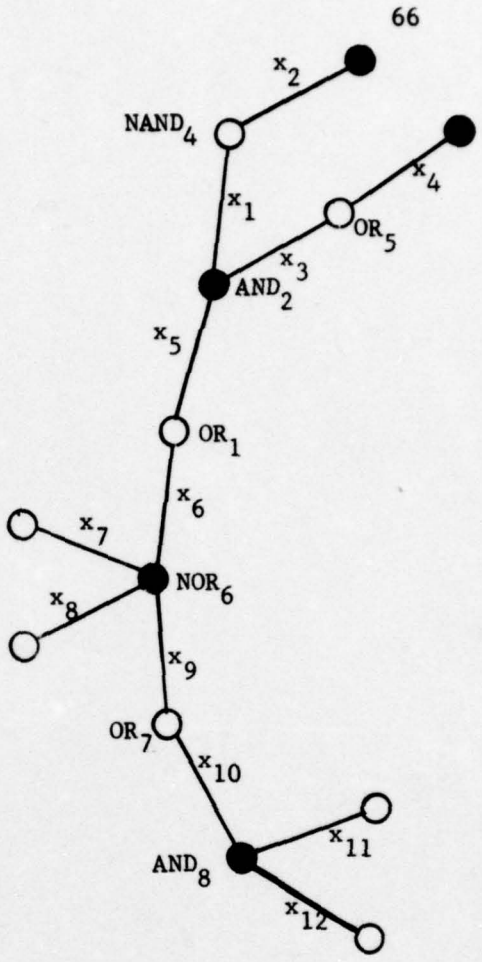


(b) level 1 added

Figure 3.10. MST construction for network of Figure 3.8.

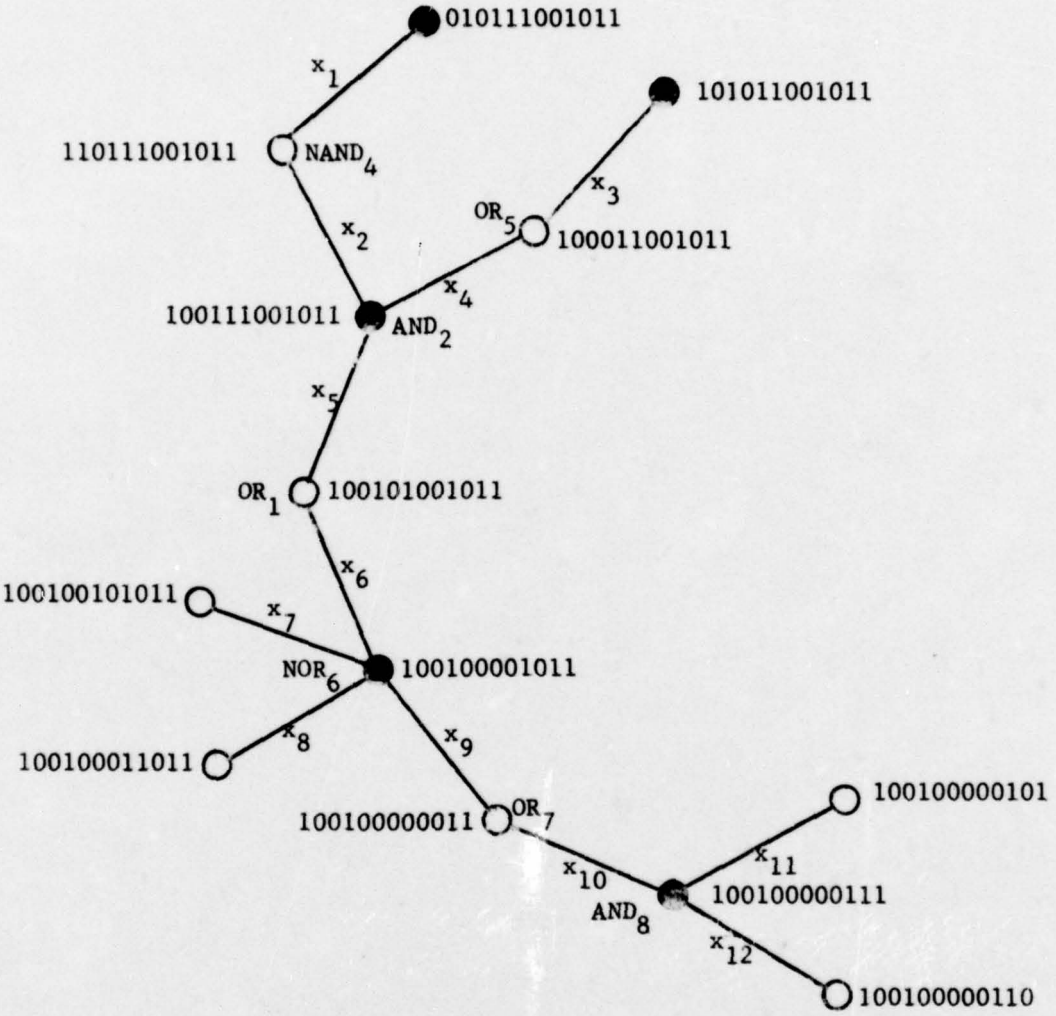


(c) level 2 added



(d) level 3 added

Figure 3.10. Continued



(e) vertices assigned

Figure 3.10. Continued

All vertices in the subtree connected to vertex NOR_6 via edge x_6 must have 100 in positions x_6 x_7 x_8 because the essential property of an MST is that the value of each variable position changes only once along any path. The same holds for the subtrees connected via edges x_7 and x_8 which have constant values of 010 and 001 respectively in x_6 x_7 x_8.

Fanout-free networks have the property that a multiple stuck-line fault involving a set of lines anywhere in the network is equivalent to a multiple fault involving only primary input lines [23,11]. (Two faults F_1 and F_2 are equivalent if $Z(X,F_1) = Z(X,F_2)$.) This property results from the fact that for any gate, a fault on the output line is equivalent to a multiple fault on the gate inputs. Each of these gate inputs which has a fault component on it is connected either a primary input or a gate output. If connected to a gate output, the fault component is again equivalent to a fault involving the inputs to that gate. This process may be continued until all fault components of the original fault have been replaced by equivalent faults involving only primary inputs. (This process may only be carried out when the network contains no fanout since a fault on a fanout branch feeding a gate is not equivalent to any fault on the fanout stem unless all fanout branches emanating from that stem are stuck at the same value.) (The interested reader may refer to (24) for a thorough treatment of fault equivalence.) Since a test which detects some fault also detects all faults equivalent to it, and since in a fanout-free network any multiple fault is equivalent to some multiple fault, all of whose components involve only primary inputs, all multiple faults anywhere in a fanout-free network

are detected by a multiple pin fault test set. Thus the preceding discussion presents a procedure for generating multiple fault test sets for fanout-free networks.

Hayes has examined the fanout structure of switching functions [25] and shown that fanout is an inherent property of switching functions. Fanout-free functions, i.e. those possessing a fanout-free realization, are studied, and a procedure for determining if a function is fanout-free is presented. This procedure could be used to determine if the preceding test generation procedure is applicable to a particular function.

The existence of MST's for other classes of functions such as unate functions has not been established however certain relationships between the prime implicants of functions are known to be necessary as stated in the following theorem.

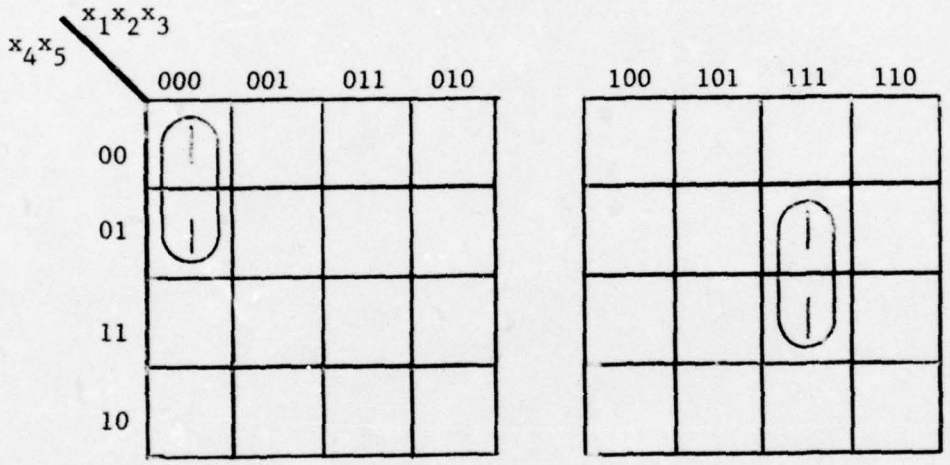
Theorem 3.7: A necessary condition for the existence of an MST in a function is that some covering set of prime implicants of the function (Definition 3.9) possess the following property: each member of the covering set is distance (Definition 3.10) ≤ 2 from at least one other member of the set.

Proof: Assume that the prime implicants of some function do not possess such a covering set. Each edge of an MST connects a true vertex to a false vertex. Consider some prime implicant. Only edges whose indices correspond to a variable which is specified in the prime implicant can emanate from that prime implicant, i.e. have the true vertex of that edge be a vertex of the implicant. Therefore, in order for all variables to be tested, there must be a path in the graph between prime implicants which are a

covering set. Since vertices along any path must be alternately true and false, the distance between any two true vertices in a path is at least 2. Therefore, since at least one prime implicant which is essential to the covering set is distance > 2 from all other members by assumption, no path can lead to a vertex in that prime implicant and therefore the variables specified only in that member of the cover cannot be tested by a tree connected to the other prime implicants. Thus an MST cannot exist. Q.E.D

Functions which do not meet the condition given in Theorem 3.7 are of course numerous, however such functions must have at least 5 variables [26]. This fact is easily demonstrated by attempting to construct a set of prime implicants which violate the condition using 4 variables. Since the prime implicants need to be separated, they need to be as small as possible, but they must have at least one unspecified variable because a prime minterm would satisfy the condition. Arbitrarily choose u000 as a prime implicant. Now a second prime implicant which covers variable x_1 must be added so that the function is nonvacuous in x_1. This may be chosen as $\alpha_1 u \alpha_3 \alpha_4$. No matter how α_1, α_3, and α_4 are chosen, the prime implicants cannot be more than distance 2 apart. With five variables, it is possible for a function to violate the condition. Figure 3.11 illustrates such a function, and it is easy to see that no MST exists for such a function.

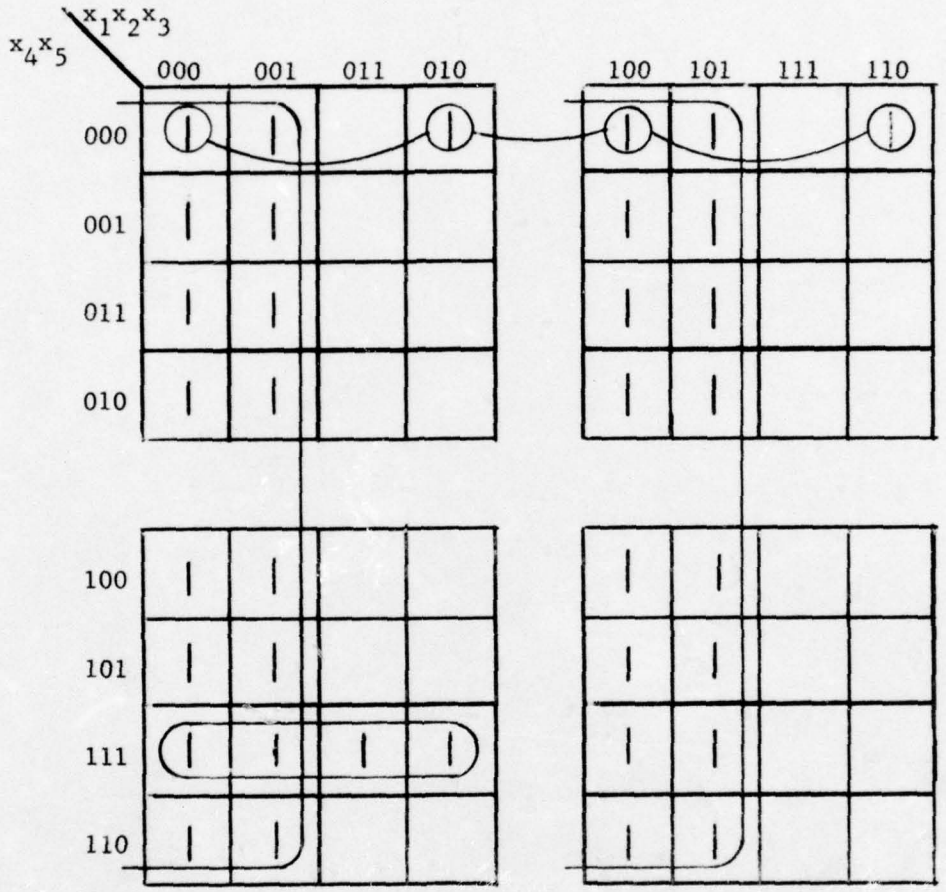
The condition of Theorem 3.7 is not strong enough to constitute sufficiency for the existence of an MST, however, as the function shown in Figure 3.12 illustrates. All these prime implicants are required to cover the 6 variables, however there can be no alternating path from uu0000 to



$$Z = \Sigma(0000u, 111ul)$$

$$= \bar{x}_1\bar{x}_2\bar{x}_3\bar{x}_4 \vee x_1x_2x_3x_5$$

Figure 3.11. A function which possesses no MST.



$Z = \Sigma(u0uuuu, uu0000, 0uu111)$

Figure 3.12. A function satisfying Theorem 3.7 which contains no MST.

0uu111 because u0uuuu only has one specified variable. Because of this, there is no way to have an edge which enters u0uuuu and a different edge which leaves u0uuuu. Thus a path must exist which goes directly from uu0000 to 0uu111. Since these two prime implicants are distance three apart, no such path exists.

3.5.2. Test Analysis Based on Growth and Disappearance of Prime Implicants

Paige [27] studied the effects which single stuck-line faults have on functions which are realized in two levels. This approach may be applied to arbitrary realizations of functions under the pin fault model. Since faults may only occur at network inputs, the faulty functions which can result under the pin fault model are closely related to the fault-free function of a network.

Consider a particular prime implicant P_k of some function. In the cubic notation used through this chapter each variable position of an implicant is specified to be either a 0 or a 1 or is unspecified (u). If input x_i is stuck at some value α this fault has one of three possible effects on prime implicant P_k ($P_{k,i}$ represents the value in the i-th position of prime implicant P_k):

1. If $P_{k,i} = u$ then P_k does not depend on the value applied to input x_i, and the prime implicant is unaffected by the fault x_i stuck-at-α.

2. If $P_{k,i} = \alpha$ then any input vertex which is adjacent to P_k in variable x_i will result in the function assuming a true output. This occurs because although an $\bar{\alpha}$ is applied at input x_i, the fault x_i stuck-at-α

causes a value of α to be transmitted to the (fault-free) network internal to the module, and the vertex seen internally is contained in P_k. The net effect of such a condition is that all such vertices become true vertices of the faulty function. This situation may be conveniently conceptualized as a growth of P_k to include these vertices.

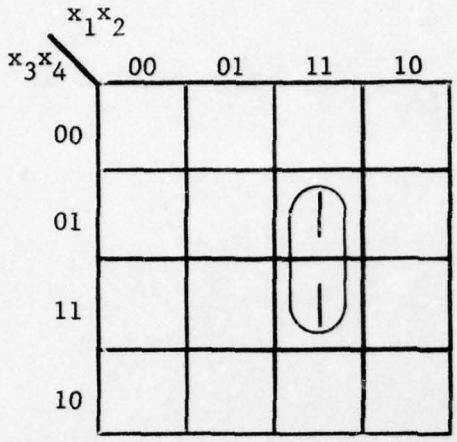
3. If $P_{k,i} = \bar{\alpha}$ then no input vertex can be applied which is contained in P_k, since the fault x_i stuck-at-α results in an α being transmitted to the internal network. The net effect which the fault x_i stuck-at-α has on prime implicant P_k may be conceputalized as the disappearance of P_k.

Figure 3.13 illustrates these three cases.

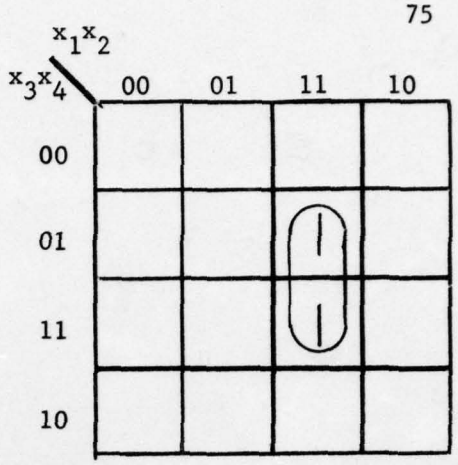
Based on this viewpoint, the faulty function which results under some multiple pin fault is determined as follows. Any prime implicant which has a specified variable with a value opposite to the polarity of a fault component on that input line disappears. Each occurrence of a specified variable which has the same value as its corresponding fault component results in a doubling of that prime implicant, i.e. the fault results in that variable becoming immaterial to inclusion in that prime implicant. Example 3.5 illustrates the determination of a faulty function from the original function and fault. A multiple pin fault for an ℓ-input function Z(x) will be denoted by an ℓ-element vector F each of whose elements is n, 0, or 1 as follows:

F_i = n $\rightarrow$ x_i is fault free (normal)

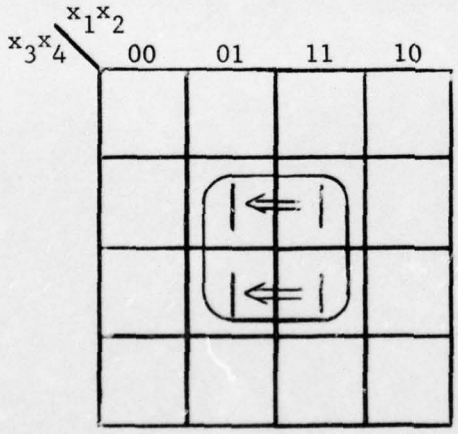
F_i = 0 $\rightarrow$ x_i is stuck at 0



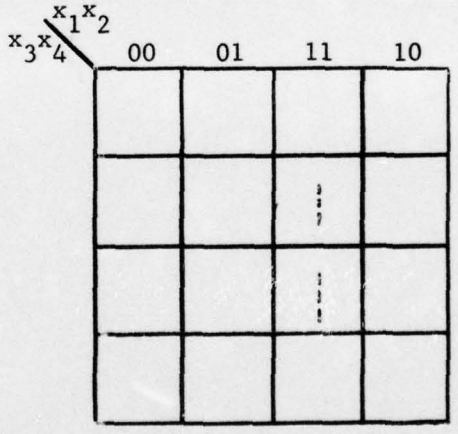
P_k = 11u1



fault: x_3 stuck-at-1
$P'_k = P_k$



fault: x_1 stuck-at-1
P'_k = ulul



fault: x_1 stuck-at-0
$P'_k = \phi$

Figure 3.13. Affects of faults on an implicant.

$F_i = 1 \rightarrow x_i$ is stuck at 1

ϕ denotes the fault free case F = (nn...n).

Example 3.5: $Z(x,\phi) = \Sigma(01luu1,\ 1u01u1,\ uu0100)$

$$F = (nn1n01)$$

$$Z(x,F) = \Sigma(01uuuu)$$

P_1 has grown in x_3 and x_6. P_2 and P_3 have disappeared.

A multiple pin fault test set may be derived in an ad hoc fashion by selecting a set of test vertices which are sufficient to insure that no growths or disappearances have occurred. For certain functions, test sets which are quite small may be derived because tests which detect many fault components simultaneously may be chosen.

A test set may be derived by a series of deductions which are based on the set of multiple pin faults which are still undetected by any previously selected test. The process terminates when no multiple pin faults remain undetected.

Definition 3.13: The fault condition of an ℓ-input combinational module relative to a partial test set is the set of assignments of values 0 (stuck-at-0), 1 (stuck-at-1), n (fault free) to the ℓ-input lines which could exist without resulting in an error indication for any test in the partial test set. Let C_τ be the fault condition relative to partial test set τ. Then $Z(t_i,f_j) = Z(t_i,\phi)$, $\forall t_i \epsilon \tau$, $f_j \epsilon C_\tau$.

A concise notation for the representation of the fault condition is a vector of ℓ sets, with each set consisting of the possible fault conditions (0, 1, or n) which may exist on each line. Any fault derived by selecting one member from each set is an element of the fault condition. The following example illustrates this notation:

If $C_\tau = (\begin{smallmatrix}0\\1\\n\end{smallmatrix}\ \begin{smallmatrix}0\\n\end{smallmatrix}\ 1\ \begin{smallmatrix}1\\n\end{smallmatrix}\ 0\ n)$ then (00110n), (nn1n0n), (1n110n), etc. are all elements of the fault condition. There are $3 \cdot 2 \cdot 1 \cdot 2 \cdot 1 \cdot 1 = 12$ such elements in this fault condition.

Note that this notation represents a Boolean expression of the following form. C_τ is true, i.e. a fault is a member of the fault condition, if and only if the first input pin is in one of the states given in the first set, and the second input pin is in one of the states given in the second set, and so forth. More precisely, C_τ is a logical product of sums expression with the further constraint that each input pin can be in only one state. The fault condition in the above example may be written explicitly as follows ($x_1/1$ means input pin x_1 stuck at 1):

$$C_\tau = (x_1/1 + x_1/0 + x_1/n)\cdot(x_2/0 + x_2/n)\cdot(x_3/1) \cdot(x_4/1 + x_4/n)\cdot(x_5/0)\cdot(x_6/n).$$

In this notation, the initial fault condition (when $\tau = \phi$) is $(\begin{smallmatrix}0\\1\\0\end{smallmatrix}\ \begin{smallmatrix}0\\1\\n\end{smallmatrix}\ \cdots\ \begin{smallmatrix}0\\1\\n\end{smallmatrix})$, and the final fault condition (when τ is a multiple pin fault test set) is (nn...n).

The fault condition for a set of test inputs may be determined by forming the intersection of the fault conditions for each of the test inputs individually. This follows from the fact that each of the individual

fault conditions must simultaneously be satisfied. This intersection operation is done on an input pin by input pin basis, and is defined by the truth table of Figure 3.14. (Whenever a null intersection results from some input, the entire fault condition is null.)

The following are examples of the intersection operation:

$$\begin{pmatrix} 0 \\ 1 \\ n \end{pmatrix} \begin{matrix} 0 & 1 & \begin{matrix} 1 \\ n \end{matrix} & \begin{matrix} 0 \\ n \end{matrix} \end{matrix} \cap \begin{pmatrix} 0 \\ n \end{pmatrix} \begin{matrix} 0 & \begin{matrix} 0 \\ 1 \\ n \end{matrix} & n & n \end{matrix} = \begin{pmatrix} 0 \\ n \end{pmatrix} \begin{matrix} 0 & 1 & n & n \end{matrix}$$

$$\begin{pmatrix} 0 \\ 1 \\ n \end{pmatrix} \begin{matrix} 0 & 1 & \begin{matrix} 1 \\ n \end{matrix} & \begin{matrix} 0 \\ n \end{matrix} \end{matrix} \cap \begin{pmatrix} 0 \\ n \end{pmatrix} \begin{matrix} n & \begin{matrix} 0 \\ 1 \\ n \end{matrix} & n & n \end{matrix} = \phi.$$

Because of the implicit logical meaning of the fault condition notation, it is quite easy to describe the event that some prime implicant has not disappeared and the event that some prime implicant has grown to cover a particular vertex or set of vertices. Other events such as the disappearance of prime implicants and nongrowth onto particular vertices are not describable because of the implicit logical meaning of the fault condition expression.

A fault condition is derived by examining the growth and disappearance of prime implicants which could result in a normal module output for a partial test. This fault condition is then refined by adding other tests to the partial test set. This process is best explained by example. Consider the function which is shown in Figure 3.11 and possesses no MST:

$$Z = \bar{x}_1\bar{x}_2\bar{x}_3\bar{x}_4 \vee x_1x_2x_3x_5$$

$$= \Sigma(P_1, P_2), \; P_1 = 0000u$$

$$P_2 = 111u1.$$

∩	0 1 n	0 n	1 n	0 1	0	1	n
0	0						
1	1	0	1	0			
n	n	n	n	1	0	1	n
0	0	0					
n	n	n	n	0	0	ϕ	n
1	1		1				
n	n	n	n	1	ϕ	1	n
0	0			0			
1	1	0	1	1	0	1	ϕ
0	0	0	ϕ	0	0	ϕ	ϕ
1	1	ϕ	1	1	ϕ	1	ϕ
n	n	n	n	ϕ	ϕ	ϕ	n

Figure 3.14. The intersection operation for fault conditions.

Select as the first test $t_1 = 00000 \in P_1$, and let Z_{UT} be the function under test which may or may not be faulty. If $Z_{UT}(00000) = 1$ then either P_1 exists and has possibly grown or P_2 has grown sufficiently to cover vertex 00000. Existence of prime implicant P_i will be denoted as P_iE, and growth of prime implicant P_i onto vertex v will be denoted as P_iGv. Fault conditions for the two possible events p_1E and p_2G (00000) are derived as follows: P_1E implies that no fault which causes the disappearance of P_1 is present, that is none of x_1 through x_4 may be stuck-at-1, hence $p_1E \rightarrow (\overset{0}{n}\ \overset{0}{n}\ \overset{0}{n}\ \overset{0}{n}\ \overset{0}{n}\ 1)$. In order for P_2 to have grown onto (00000), fault components sufficient to transform vertex (00000) into a vertex in P_2 must be present, hence $P_2Gt_1 \rightarrow (1\ 1\ 1\ \overset{0}{n}\ 1\ 1)$. Since one of these two fault conditions must exist for $Z_{UT}(t_1) = 1$, the following expression may be written:

$$Z_{UT}(t_1) = 1 \rightarrow p_1E \vee p_2Gt_1$$

$$\rightarrow (\overset{0}{n}\ \overset{0}{n}\ \overset{0}{n}\ \overset{0}{n}\ \overset{0}{n}\ 1) \vee (1\ 1\ 1\ \overset{0}{n}\ 1\ 1).$$

Similarly, if $t_2 = 11111 \in P_2$ then

$$Z_{UT}(t_2) = 1 \rightarrow p_2E \vee p_1Gt_2$$

$$\rightarrow (\overset{1}{n}\ \overset{1}{n}\ \overset{1}{n}\ \overset{0}{n}\ \overset{1}{n}\ 1) \vee (0\ 0\ 0\ 0\ \overset{0}{n}\ 1).$$

These two fault condition expressions may now be combined as follows:

$$Z_{UT}(t_1,t_2) = (1,1) \rightarrow (p_1E \vee p_2Gt_1) \cap (p_2E \vee p_1Gt_2)$$

$$\rightarrow [(\overset{0}{n}\ \overset{0}{n}\ \overset{0}{n}\ \overset{0}{n}\ \overset{0}{n}\ 1) \vee (1\ 1\ 1\ \overset{0}{n}\ 1\ 1)] \cap [(\overset{1}{n}\ \overset{1}{n}\ \overset{1}{n}\ \overset{0}{n}\ \overset{1}{n}\ 1) \vee (0\ 0\ 0\ 0\ \overset{0}{n}\ 1)]$$

$$\rightarrow (n\ n\ n\ \overset{0}{n}\ \overset{1}{n}) \vee (0\ 0\ 0\ 0\ \overset{0}{n}\ 1) \vee (1\ 1\ \overset{0}{n}\ 1\ 1).$$

From the above expression it is apparent that tests t_1 and t_2 have detected nearly all of the 3^6-1 multiple pin faults. Notice that the last two terms in the above fault condition expression correspond to growth of P_1 over the entire space and growth of P_2 over the entire space respectively. Thus all of the faulty functions represented by these two terms are tautologies and will be detected by any test which has a normal output of 0. Since any test set must have at least one such test to detect stuck-at-1 on the output pin, these two terms will eventually be eliminated. This leaves only three faults undetected, namely (nnn0n), (nnnn1), and (nnn01). The fault (nnn0n) corresponds to a growth of P_1 along x_4 and will be detected by either 00011 or 00010, independent of the fault state of line x_5. The fault (nnnn1) is detected by checking for growth of P_2 into 11100 or 11110 and is independent of the fault state of line x_4. Consequently only two more tests are required to complete the multiple pin fault test set. Hence the following is a 4 element test for the function Z which possesses no MST:

$$T = \{00000, 11111, 00011, 11100\}.$$

The growth and disappearance approach is not generally amenable to test set generation although it is useful in the analysis of tests. The applicability of this type of analysis is highly dependent upon the prime implicant structure of the particular function. Functions with few prime implicants which are relatively distant from each other are easily analyzed because multiple pin faults of high multiplicity are required in order for the growth of one prime implicant to mask the disappearance of

another. Functions possessing no MST are generally of this type, and all such functions which have been studied may be shown to possess test sets of size $\leq$ n+1 by use of the techniques described in this section. Unfortunately functions of this type have eluded classification, and a general theory of test generation based on growth and disappearance testing has not been developed. This section is included in order to present an approach to fault and test analysis which may be further developed in the future, and to illustrate that test sets of small size which do not satisfy Theorem 3.1 do exist for certain functions.

3.6. Multiple-Output Functions

The results of this chapter extend in a straightforward manner to modules with multiple output pins. Such a module is depicted in Figure 3.15. Each of the output pins Z_1 through Z_m in Figure 3.13 realizes a combinational function which depends nonvacuously on some subset of the input pins x_1 through x_ℓ.

Definition 3.14: The input set for each output pin Z_j is the set of input pins carrying the input variables upon which the function appearing at output Z_j depends nonvacuously. This set is denoted by I_j.

A common practice in integrated circuit design is to replicate a function or to implement more than one function on a single chip when chip area and package pins are available. The result is that an integrated circuit module may have outputs or sets of outputs whose input sets are

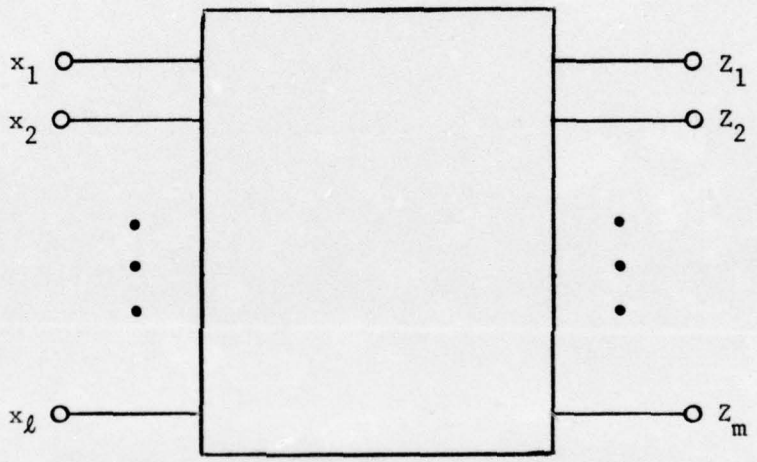


Figure 3.15. A multiple-output combinational module.

disjoint from the input sets of the remaining outputs. This property of disjoint input sets may be used to partition a module into submodules which are independent of each other. Fault detection tests may then be generated for the module by generating tests for each submoduel. A submodule which has one output is handled as if it were a single output module, even though it shares a physical package with other submodules. This section deals with the case in which a module or submodule has more than one output, and each of the input sets for these outputs has a nonnull intersection with the union of the input sets of the other outputs. The term "module" will be used to refer to such a module or submodule in the remainder of this section.

The alternating adjacency concept used in generating tests for single-output modules may be extended to the multiple-output case. An input may be tested by applying a pair of test vertices which differ only in the value applied to that input such that the value appearing at some output changes in the fault-free circuit. Since such a pair of tests normally causes both values to appear on some output, that output is also tested for both stuck-at-1 and stuck-at-0 faults. A multiple pin fault test set may therefore be formed by determining a set of input vertices which contain an alternating adjacency for each input pin and has the further property that each output pin is used to monitor at least one of the alternating adjacencies. It is therefore possible to test a multiple-output module without having to add tests beyond those required for testing the inputs. A test set which tests all inputs but not all of the outputs

may be augmented to test the untested outputs by adding at most one additional test input for each untested output, and it may be possible to determine a single input vertex or a small number of vertices which cause all of the untested outputs to assume values complementary to the output values produced under the original partial test set. It may also be possible to test several inputs with one pair of input vertices. For example if $I_1 \not\subset I_2$ and $I_2 \not\subset I_1$ then it may be possible to test an input contained in $I_1 \cap \bar{I}_2$ and simultaneously test an input contained in $I_2 \cap \bar{I}_1$. This is possible whenever the values applied to the inputs contained in $I_1 \cap I_2$ allow both an input in I_1 and an input in I_2 to simultaneously be sensitized to Z_1 and Z_2 respectively.

The algorithms previously developed for the single-output case may be used to generate a multiple pin fault test set for multiple-output modules by the following procedure:

Step 1: Form a subset, Z^*, of the outputs which has the property that

$$\bigcup_{\forall i \ni Z_i \in Z^*} I_i = \{x_1 x_2 \ldots x_\ell\}.$$

Step 2: For each element $Z_j \in Z^*$ examine I_j. For each input x_i contained in I_j for which tests have not yet been generated, use the function specification for Z_j as input to the single-output test generation algorithm to generate a pair of tests for input x_i. Repeat this step until tests have been generated for $x_1 \ldots x_\ell$. Leave all input variables noncritical to the test pair unspecified.

Step 3: For each output $Z_k \notin Z^*$ determine the fault-free circuit response to the tests generated in Step 2. Outputs which produce both 0 and 1 values for this set of tests need not be considered further. Select values for the as yet unspecified input variables to cause both values to appear on the remaining outputs wherever possible. Each output which produces only 0's or 1's after this has been completed requires an additional test selected to produce a 1 or 0 respectively. The set of tests produced in steps 2 and 3 constitutes a multiple pin fault test set for the module.

There are many possibilities for heuristic improvement to the above procedure such as selecting Z^* to reduce the amount of computation required to test the inputs, or if $|\{Z\}| \leq \ell$ using $Z^* = Z$ and selecting inputs such that Step 3 is eliminated. It may even be feasible to consider multiple-output modules on an individual basis and adapt the test generation procedure accordingly.

3.7. Summary

This chapter has investigated fault detection under the multiple pin fault assumption for combinational modules. Test sets with the property that all single fault components are detected completely have been studied, and a method of generating such test sets developed. The test sets produced by the main algorithm presented have the property that exactly half of the test inputs are true vertices. It is therefore possible to apply these test inputs in an order which results in a strictly alternating output

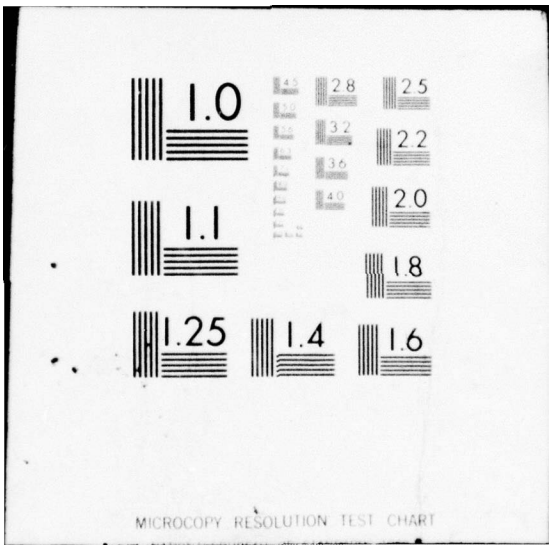
sequence, thus eliminating the need to store the normal output response to the test set. The computational complexity of generating test sets was examined, and shown to be within the realm of feasibility. Other types of test sets were also examined, and the extension to multiple-output modules considered.

AD-A031 431 ILLINOIS UNIV AT URBANA-CHAMPAIGN COORDINATED SCIENCE LAB F/G 9/2
AN INTEGRATED CIRCUIT FAULT MODEL FOR DIGITAL SYSTEMS.(U)
SEP 76 M L KETELSEN DAAB07-72-C-0259
UNCLASSIFIED R-743 NL

2 OF 2
AD
A031431

END

DATE
FILMED
11-76


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
MICROCOPY RESOLUTION TEST CHART

4. FAULT DETECTION IN SEQUENTIAL CIRCUITS

4.1. Introduction

In this section, fault detection under the pin fault assumption is examined for circuits which exhibit sequential behavior. Figure 4.1 illustrates the general model which is used to describe sequential circuits. In Figure 4.1 the elements marked D represent signal delays which act as memory capable of storing the internal state of the machine. Both synchronous and asynchronous modes of operation will be considered in this chapter.

In the abstract model of Figure 4.1, the synchronous mode of operation is characterized by the fact that inputs occur at discrete intervals of time, and that each application of the set of inputs results in at most one state transition. Since the outputs are functions of the inputs, they are also valid only during the time interval in which the inputs are applied. In order to assure valid operation of such a scheme, it is necessary to restrict the circuit delays and the delays of the "D" elements so that state variables appearing on lines $y_1 \ldots y_k$ do not change values until after the completion of the interval during which the inputs are applied. In actual practice the problems associated with synchronous operation are usually solved by using bistable logic elements (flip-flops) instead of delays for retention of the state information. A special synchronizing input called the "clock" is used to gate the excitation lines $Y_1 \ldots Y_k$ into the flip-flops. In the remainder of this chapter, it will be assumed that synchronous circuits are implemented using suitable flip-flops (master-slave or edge

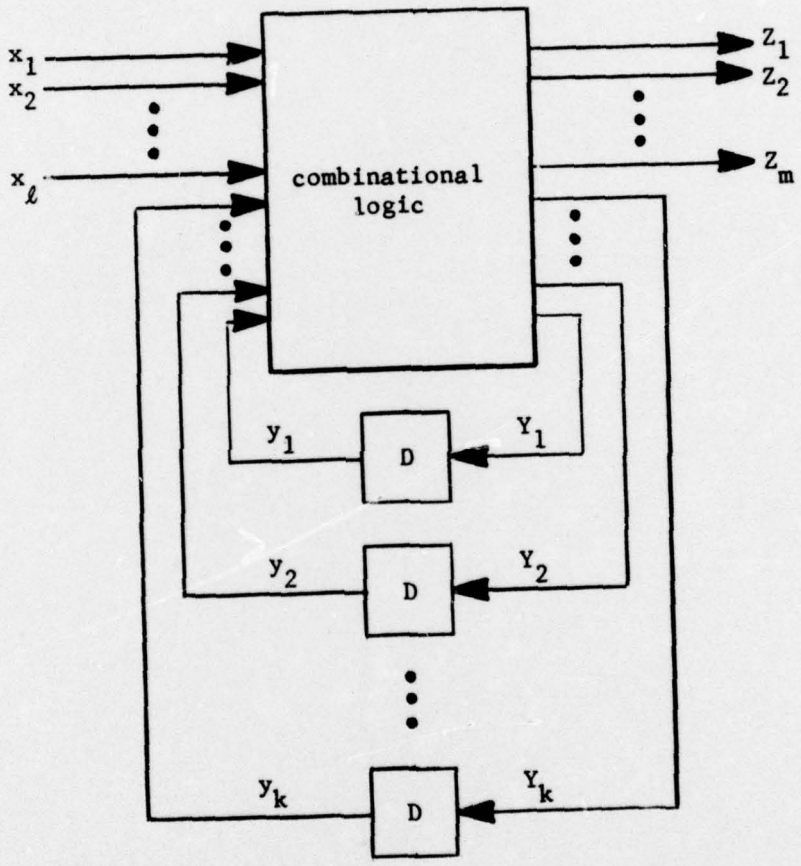


Figure 4.1. Sequential machine.

triggered) and that a clock input is used to synchronize the state transitions.

In the asynchronous case there is no clock input, and the circuit responds directly to changes in the values on the input lines. Frequently the delay elements shown in Figure 4.1 are not implemented explicitly, but rather the inherent gate delays in the combinational logic network suffice to store the state information until a stable state, i.e. when $y_i = Y_i$ $1 \leq i \leq k$, is achieved. Alternately, unclocked set-reset flip-flops may be used. Although synchronous circuits may be viewed as asynchronous circuits by treating the clock input as one of the primary input lines, these two modes of operation will be discussed separately.

In the discussion throughout this chapter it is assumed that all pin faults are detectable, i.e. that no input or output lines are redundant, and that all flow tables are reduced and strongly connected. Asynchronous machines are assumed to operate in the fundamental mode.

The nature of the fault detection problem for sequential circuits is quite different from that for combinational circuits because of the fact that the output of such a circuit is, in general, a function of previously applied inputs in addition to the current input. Some of the inputs ($y_1 \ldots y_k$) to the combinational logic are not directly controllable, and some of the outputs ($Y_1 \ldots Y_k$) are not directly observable. As a result, if it is desired to apply some particular input pattern to the combinational logic portion of the circuit, it is first necessary to apply a sequence of inputs which result in the desired values appearing on $y_1 \ldots y_k$ and then apply the desired values to the directly accessible lines $x_1 \ldots x_\ell$.

In order to observe the output of the combinational logic to such an input pattern, it is then necessary to apply a sequence which will lead to an indirect display at $Z_1 \ldots Z_m$ of the values which appeared at the outputs $Y_1 \ldots Y_k$ as a result of the applied input pattern. The fault detection problem for sequential circuits is to determine a sequence of input patterns which when applied to a circuit under test will result in an output sequence which is correct only if the circuit under test is fault-free under the pin fault assumption.

4.2. Fault Detection Sequences for Synchronous Circuits

This section discusses the generation of fault detection sequences for synchronous circuits under the pin fault assumption. Throughout this section, the Mealy machine formulation will be used. In the Mealy machine, the current output is a function of both the current internal state $(y_1 \ldots y_k)$ and the current input. Such machines are represented by flow tables in which each entry, which corresponds to a particular current state and input, gives the next state and the output. Figure 4.3a (p. 95) illustrates such a flow table.

Since this thesis is concerned with the derivation of fault detection tests for existing circuits, discussion will be restricted to flow tables in which all flow table entries are specified, unless otherwise noted. Even if the specification for a particular sequential function is incomplete, as is quite often the case, every realization of the function has a completely specified flow table.

4.2.1. The Effects of Pin Faults on Synchronous Circuits

The locations of fault sites under the pin fault assumption is of course dependent upon the manner in which the circuit is partitioned into integrated circuit modules. In this section, we will assume that faults may occur at the primary input lines $x_1 \ldots x_\ell$, the primary output lines $z_1 \ldots z_m$, the "next state" feedback lines $Y_1 \ldots Y_k$, the state variable lines $y_1 \ldots y_k$, and all lines connected to the clock input. This assumption is quite broad in the types of partitioning which are included. Figure 4.2 illustrates such a partitioning. Any partitioning which does not contain any multiple-module signal paths in the combinational logic portion of the circuit complies with this assumption. Additionally if some or all of the memory devices and feedback lines are internal to some of the modules, this configuration is covered by the above assumption.

Within these topological limits, the effects that pin faults can have on circuit behavior may now be examined. It is assumed that the networks to be considered implement strongly connected, reduced flow tables, and that the inputs and outputs are irredundant in the sense that no input or output line may be replaced by a line fed with a constant logical value of 0 or 1. First of all, if the clock input to some module has failed to either 0 or 1, then the flip-flops which are controlled by that clock input will never change state. Hence such a fault will be equivalent to a multiple stuck fault on some subset of the state variable lines. Any faults on either excitation lines ($Y_1 \ldots Y_k$) or state variable lines ($y_1 \ldots y_k$) will result in a faulty machine with fewer states than the original machine. If all excitation, state variable, and clock lines are

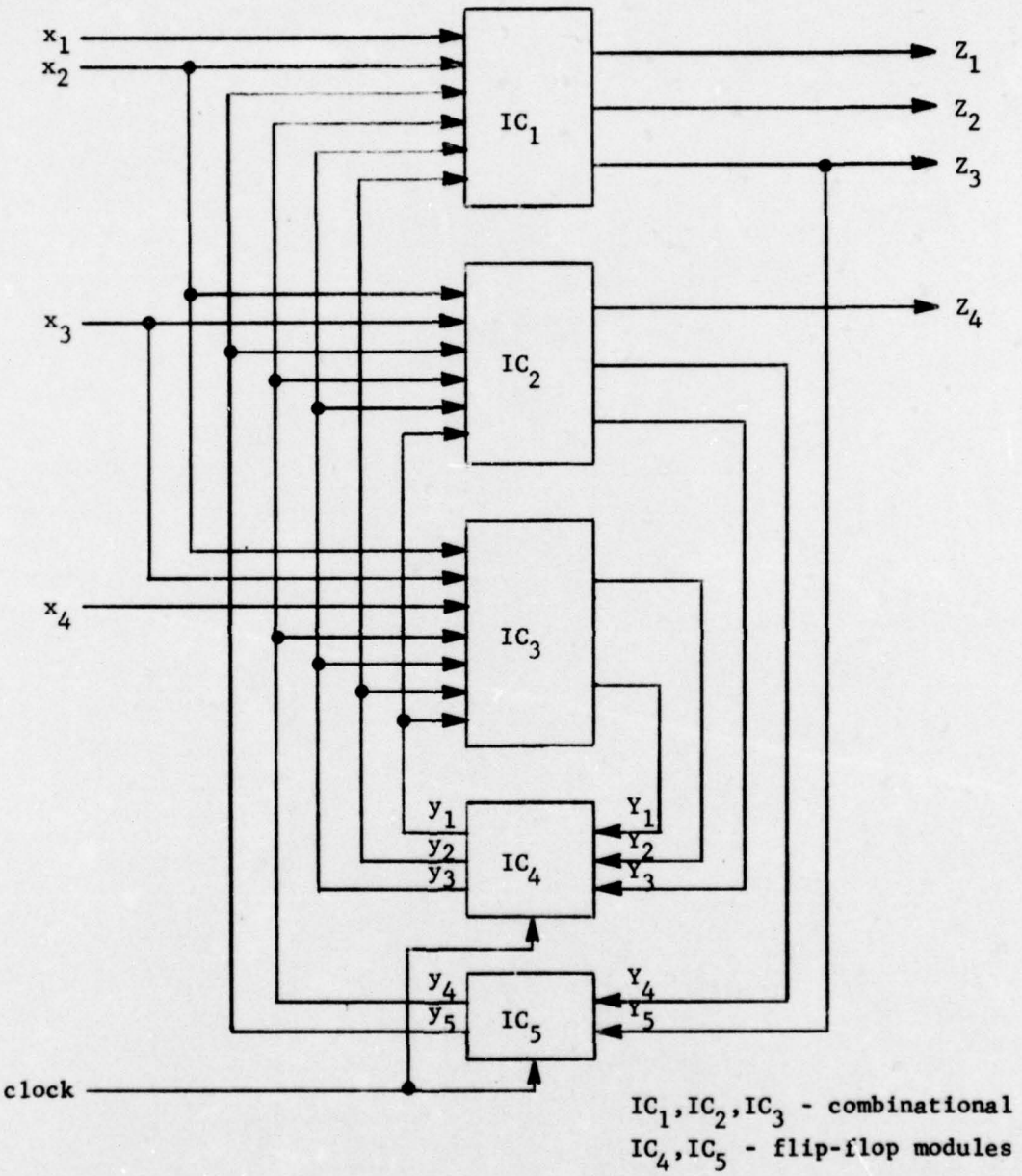


Figure 4.2. Example of partition of synchronous circuit into integrated circuit modules.

fault-free then the only possible fault sites are the primary inputs $x_1 \ldots x_\ell$ and the primary outputs $z_1 \ldots z_m$. The effect of faults on output pins is obvious and is tested by any sequence which results in a normal output sequence requiring each output to assume both a 0 and a 1 somewhere in the sequence. The effect that faulty input pins have on circuit behavior is that constant values are transmitted to the logic network internal to the integrated circuit module regardless of what value is applied at the inputs to the circuit. As a result, certain input columns in the flow table are never entered. For example if a three-input circuit has the fault x_2 stuck-at-1 then input columns of the flow table which have $x_2 = 0$ will never be entered by the machine. The next state and output for a given input will be determined by the input values received by the logic internal to the integrated circuit modules. As a result, the effect which input pin faults have on the flow table is to cause columns to be replaced by columns which are adjacent in the faulty input variables. Figure 4.3 illustrates this transformation of the flow table. In this figure, the fault x_1 stuck-at-1 results in column 00 being replaced by the entries found in column 10, and column 01 being replaced by the entries in column 11. If both x_1 and x_2 were faulty, then all columns would be the same as the column corresponding to the double fault. It can be seen from Figure 4.3 that input pin faults may or may not directly affect the number of states.

As the above discussion indicates, faults of the pin fault type result in a very restricted and well-defined corruption of the normal machine. Furthermore, this faulty circuit behavior is completely

Present state \ x_1x_2	Next state, output 00	01	11	10
A	B,0	A,0	C,1	B,1
B	C,0	D,0	C,1	C,1
C	C,1	D,1	D,1	D,0
D	C,1	D,1	A,1	A,0

(a) Normal machine.

Present state \ x_1x_2	00	01	11	10
A	B,1	C,1	C,1	B,1
B	C,1	C,1	C,1	C,1
C	D,0	D,1	D,1	D,0
D	A,0	A,1	A,1	A,0

(b) Faulty machine resulting from fault x_1 stuck at 1.

Figure 4.3. Transformation of flow table under input pin fault.

characterized by modifications to the fault-free flow table, independent of any specific implementation. This is a significant improvement over the situation which exists for synchronous machine diagnosis under previously studied fault assumptions. In the machine identification approach [28,29] it is assumed that faults may result in arbitrary sequential behavior as long as the number of states does not increase. This assumption admits a large number of faulty machines which could not result from any physical failure. As a result of such a "loose" fault assumption, testing sequences are extremely difficult to generate, and are very long. The circuit testing approach [30,10], based on a particular implementation in the context of a specific fault model, results in shorter testing sequences. However, when the fault model used here is applied, the circuit testing approach becomes independent of the circuit implementation and in addition short test sequences may be readily determined.

The discussion which follows deals with the generation of test sequences for synchronous circuits. In this discussion, attention is focused on pin faults which occur at the primary inputs. It will be shown that clock pin failures will automatically be checked, and that any output pins which are not tested are easily discovered and trivially tested by concatenating a sequence which causes the machine to exercise untested output pins. Since multiple pin faults are assumed the fanout structure of a particular integrated circuit implementation does not affect test generation.

4.2.2. Minimum-Length Fault Detection Sequences

It is always possible, at least in principle, to determine a minimum-length fault detection sequence by exhaustive methods. This of course applies for any fault model, but it is necessary to examine all possible faulty machines in addition to all possible input sequences of length less than the minimum length. Such an approach is not practical in general; however, under the pin fault model a relatively restricted set of faulty machines results, and for machines with a small number of inputs, this approach may be employed if there is sufficient motivation for minimizing the length of the test sequence. A method for systematically determining the minimum length pin fault detection sequence will be described.

This exhaustive process begins by first determining the flow tables for all adulterated machines which may result due to input pin faults, as described in the previous section. Each of the states in this set of flow tables is then assigned a unique label. A circuit which is to be tested is in one of the states of the set of faulty flow tables or in one of the states of the normal flow table if no faults are present. A successor tree [31] may now be constructed for the "direct sum" machine resulting from the concatenation of the set of faulty flow tables with the normal flow table. The initial state uncertainty is total, since there is no knowledge of the fault condition or initial state of the circuit under test. Observation of the output which results from application of input symbols allows this initial uncertainty to be reduced with the application of each input symbol. For example, if the machine may be in any one of the four states A, B, C, or D, then the initial uncertainty is (ABCD). If an

input I_1 is applied and $Z(A,I_1) = Z(B,I_1) = 1$ and $Z(C,I_1) = Z(D,I_1) = 0$ then after the application of I_1 and observation of the output Z, the uncertainty is (AB)(CD). Rather than recording the states which the machine could have been in, it is more convenient to record the current state uncertainties at each node. Thus, the elements of the various uncertainties are not necessarily distinct. The successor tree displays the successor uncertainties for all inputs. Obviously it is unnecessary to continue the tree from nodes which are identical to some other node, since the subtree will be the same as some other portion of the tree. A fault detection sequence is indicated when some node of the successor tree has the property that any uncertainty containing a state which is a state of the normal machine contains no states which are states of faulty machines. This property assures that a correct output response to the input sequence corresponding to the path from the initial uncertainty to such a node will guarantee that the machine is not in any state which is a state of one of the faulty machines. If one desires to know precisely what fault is present, this procedure may be continued until a node is found which has the property that each uncertainty vector contains states belonging to only one faulty machine. Figure 4.4 illustrates the construction of a minimum length pin fault detection sequence. Since this particular machine possesses no distinguishing sequence, diagnosis under the general fault assumption of Hennie results in an extremely long sequence (173 inputs!), compared to 6 inputs under the pin fault assumption.

Present State \ x	0	1
A	B,0	D,0
B	A,0	B,0
C	D,1	A,0
D	D,1	C,0

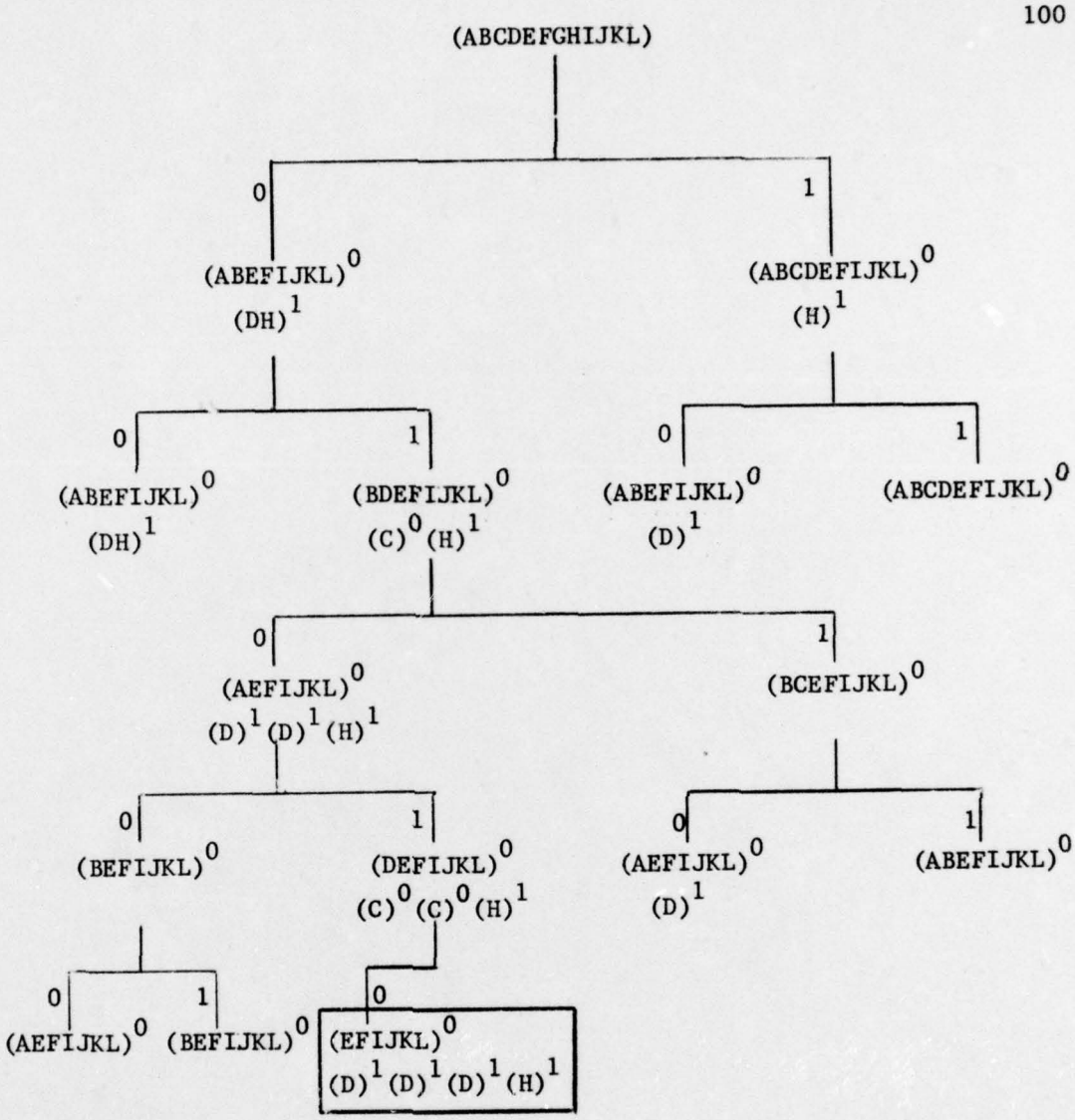
next state, output

(a) Normal machine.

	Present state \ x	0	1
normal	A	B,0	D,0
	B	A,0	B,0
	C	D,1	A,0
	D	D,1	C,0
x stuck at 0	E	F,0	F,0
	F	E,0	E,0
	G	H,1	H,1
	H	H,1	H,1
x stuck at 1	I	L,0	L,0
	J	J,0	J,0
	K	I,0	I,0
	L	K,0	K,0

(b) Direct sum machine for input faults.

Figure 4.4. Minimum length test sequence generation.



pin fault detection sequence: 01010
valid responses: 00001
00101
10101

(c) Successor tree

Figure 4.4. continued

4.2.3. Generation of Pin Fault Detection Sequences

This section deals with the generation of pin fault detection sequences. Such sequences will be constructed by concatenating subsequences which have special properties. The following definitions are useful in the discussion of such sequences (in these definitions, machine M is fault-free).

Definition 4.1: A homing sequence for machine M is an input sequence which when applied to M allows the final state to be uniquely determined through observation of the resulting output sequence, regardless of the initial state.

Definition 4.2: A synchronizing sequence for machine M is an input sequence which leads M to a specified final state, regardless of the initial state.

Definition 4.3: A distinguishing sequence for machine M is an input sequence which has the property that the resulting output sequence is unique for each possible initial state.

Definition 4.4: A transfer sequence from state Q_i to state Q_j is a sequence which when applied to machine M in state Q_i leads M to state Q_j. Such a sequence is denoted $T(Q_i,Q_j)$.

The following shorthand notation for specific input faults will be used. The fault "input x_i stuck-at-d," $d \in (0,1)$, will be denoted x_i/d.

As previously discussed, faults occurring at primary inputs to the machine render certain columns of the fault-free flow table unreachable. Faults at the inputs to the memory elements, outputs of the memory elements, and feedback inputs of the combinational logic result in a reduction in the

number of states. Faults on output pins render these lines inactive. In this section a general method for designing sequences which detect these faults will be formulated.

The fault detection sequence to be derived consists of three portions. The first part serves the purpose of verifying that the machine under test possesses the proper number of states, and also establishes the validity of certain transfer sequences. This part of the test therefore tests for pin faults on lines into and out of the memory elements. The second part of the fault detection sequence checks certain key transitions, the validity of which guarantees that no input lines have failed. In order to assure that no output lines have failed it is necessary for the fault-free response to the fault detection experiment to possess values which place both 0 and 1 on each output line. The third part of the fault detection sequence consists of an input sequence which exercises any previously untested output lines.

The first part of the fault detection sequence is the same as that developed by Hennie and adapted by Kohavi. The sequence is initiated by a sequence which brings the fault-free machine to a known state. This may be accomplished by a synchronizing sequence if one exists, or by a homing sequence followed by an adaptively selected transfer sequence. For machines which possess a distinguishing sequence, the number of states may be verified by applying a distinguishing sequence followed by a transfer sequence which brings the machine to a new state, then another distinguishing sequence, and so on. For a machine with N states, this portion of the test is completed by applying such a sequence which results in N different

responses to the distinguishing sequence. The intermediate transfer sequences used in this part of the test are also verified since the terminal state of the transfer is checked. For machines which do not have a distinguishing sequence, it is necessary to use locating sequences and characterizing sequences in order to establish the existence of N distinct states, as described in [10]. (Such machines will not be considered in detail here because they are usually avoided in actual practice due to the difficulty of diagnosis and because the rather lengthy state identification problem is considered elsewhere.)

Consider the machine whose flow table appears in Figure 4.3a. This machine has a synchronizing sequence (00,00) which leads the machine to state C, and a distinguishing sequence (10,10), Thus the first part of the fault detection sequence consists of the following sequence:

Input (x_1x_2)	00	00	10	10	10	10	10	10
State	?	?	C	D	A	B	C	D	A
Output			0	0	1	1	0	0

In this case it is possible to overlap the various applications of the distinguishing sequence, and it is unnecessary to use any intermediate transfer sequences since the machine is left in a different state after each input is applied. The fact that four different responses to the sequence (10,10) are observed for the fault-free machine means that a machine under test which responds correctly to this sequence has four

states and also that T(C,D) = (10), T(D,A) = (10), and T(A,B) = (10). Note that one additional application of the distinguishing sequence determines which state is entered after state B. This verifies the transfer sequence T(B,C). Since the machine has 4 states, the final state which is verified, B in this case, must return to one of the previously encountered states. By adding this extra distinguishing sequence, this previous state is determined. The significance of this (N+1)st application of the distinguishing sequence is that it indicates which state the machine is in at the conclusion of this part of the test, and it verifies a transfer sequence from the last state verified to one of the others. With this transfer sequence plus the others, it is possible to get from any state to any other using only transfer sequences which have been verified.

The second part of the fault-detection test sequence tests a set of specific transitions which guarantee the absence of input faults. It is first necessary to deduce such a set. If the machine is brought to a known (in terms of its response to the distinguishing sequence) state and an input is applied, there are two types of errors which can result. The machine can be transferred to an erroneous state or the machine can produce an erroneous output. If the output is correct then an application of the distinguishing sequence can be used to check the new state. If an erroneous output is produced for the faulty transitions of interest, then a distinguishing sequence is unnecessary. The fault detection capabilities of a particular transition may be deduced by examining the other transitions which share that flow table row. For example in the machine of Figure 4.3a, consider the transition from state C under input 01. An input fault of

$x_2/0$ or $x_1/0$, $x_2/0$ results in a transition to state C with an output of 1. Therefore these faults can be detected by application of the distinguishing sequence. The fault $x_1/1$, $x_2/0$ results in a transition to state D, the proper state, but the output is erroneously a 0. Hence this fault will be detected immediately. The faults $x_1/1$ and $x_1/1$, $x_2/1$ may not be detected by this particular transition, because both the next state and the output which occur in the presence of these faults are correct. Using this type of reasoning, fault condition expressions as defined for combinational circuits may be written and a set of transitions which leads to a fault condition of (nn...n) constitutes a set of transitions sufficient to diagnose the circuit. The transitions which require a distinguishing sequence in order to detect the desired faults must be noted. In the example of Figure 4.3a, this process is as follows: Rows C and D have the property that under input 10, the output is a 0 but every other transition in these rows produces a 1. Hence all faults which are detectable with these two transitions are detected immediately without the necessity for an application of a distinguishing sequence. Suppose the transition from state C under 10 is selected, then all faults other than $\binom{1\ 0}{n\ n}$ will be detected. The transition from state A under input 01 leads to a state which is different than that under any of the other columns, and in particular from columns which could be erroneously entered if any of the faults in $\binom{1\ 0}{n\ n}$ are present. The fault condition for this transition is $\binom{0\ 1}{n\ n}$. Since $\binom{1\ 0}{n\ n} \cap \binom{0\ 1}{n\ n} = (n\ n)$, these two transitions are sufficient to detect all multiple input faults. The transition from state A under input 01 must be followed by a distinguishing sequence since one of the faults untested

by the first transition, $x_2/0$, can cause the machine to produce the correct output. The second part of the fault detection sequence may now be constructed using the transition set {C(10),A(01)}, remembering that the first transition does not require a distinguishing sequence. Since this particular machine has only one output line, the third part of the test sequence is not required. The following fault detection sequence results:

		part 1								part 2		
Input (x_1x_2)		00	00	10	10	10	10	10	10	01	10	10
normal response	State	?	?	C	D	A	B	C	D	A	A	B	C
	Output				0	0	1	1	0	0	0	1	1
									*			*	
									C(10) checked			A(01) checked	

The example above was used by Kohavi to compare his method which considered a specific implementation and the single fault assumption to the method of Hennie. Part 1 is the same for all three methods. The remainder of the sequence derived by Kohavi required 15 inputs compared to 41 inputs required for Hennie's method. Thus it is seen that a significant reduction in the length of fault detection sequences as well as a simplified generation procedure results from the pin fault assumption in the case of synchronous circuits.

4.3. Fault Detection Sequences for Asynchronous Circuits

Fault detection in asynchronous circuits has received relatively little attention in the past. A primary reason for this is that under previously considered fault models, it is possible for critical races to result from certain faults. If this occurs the behavior of the faulty machine may be nondeterministic. In order to circumvent such difficulties, it is necessary to place restrictions on the structure of the fault-free machine. Under the pin fault assumption, it is not possible for a race-free circuit to be transformed by a fault into one containing races. There are other difficulties in testing asynchronous circuits. Since asynchronous circuits respond to input changes, it is not possible to repeat an input without applying an intermediate sequence. The particular implementation technique may also place restrictions on the types of input sequences which may be applied, for example the circuit may have been designed under the assumption of single input changes only, and therefore a fault detection sequence must be designed which complies with this restriction. Another inherent difference between synchronous and asynchronous machines is that in the synchronous case, the state of the machine is determined by the values taken on by the internal state variables, whereas in the asynchronous case, the state is determined by the values on the internal state variables in addition to the values on the input lines. This state is referred to as the "total state" of the machine. A distinguishing sequence for an asynchronous flow table must be capable of identifying each of these total states, and in general there will be many more total states than there are internal states. The distinguishing sequence must also satisfy any input

restrictions. It is therefore necessary to assume that the flow table is reduced under the specific input restrictions.

The behavior of such circuits in the presence of faults of the pin fault type is similar to that described in Section 4.2.1. As in the synchronous case, input faults result in an adulterated flow table in which a column which is unreachable due to the fault is replaced by the column which is erroneously entered. However if a fault is present which masks a particular input change to the internal logic, no state transition occurs, whereas some state transition always occurs for each cycle of the clock input in the synchronous case. Since asynchronous circuits are generally implemented by relying on propagation delays of the logic elements for the storage required to maintain the current internal state until the new stable state is reached, it is much more likely that state variable lines will be retained within integrated circuit boundaries rather than being routed externally from output pins to input pins. If this is true, then the internal state variables may be assumed to be fault-free. If an asynchronous machine possesses external feedback, then faults at either end of the routing result in a reduction of the number of internal states, and therefore the number of total states. Since the internal logic of the machine may be assumed fault-free, complete diagnosis results from a verification of all input and output lines and a verification of any feedback lines which possess potential fault sites. This guarantees the existence of all of the total states without the necessity for explicitly testing each of them.

In the next two sections, the design of fault detection sequences will be considered. This design process is greatly enhanced by the presence

of certain flow table properties. In Section 4.3.1 these properties will be identified by means of example, and a methodology for fault detection sequences, based on these properties, will be developed. Section 4.3.2 considers diagnosis of machines which lack these properties.

4.3.1. Fault Detection Sequences for Unit Transition Diagnosable Machines

The nature of asynchronous behavior, namely that a state transition occurs only when the internal logic senses a change in the input state, greatly simplifies the test sequence generation process. To illustrate this simplification, consider the following hypothetical input sequence and corresponding output sequence (the times shown indicate when the inputs changed, and are arbitrary intervals):

time:	0	1	2	3	4	5	6	7
input ($x_1x_2x_3$):	000	001	011	001	101	111	011	001
output z:	1	0	0	0	1	1	0	1

Assume that the above input sequence is applied to a three-input single-output asynchronous machine which has internal feedback lines, and that the above output sequence results. It may be concluded from this sequence alone that the machine is pin fault-free, without even considering the flow table of the machine. The reasoning behind this conclusion is as follows: At time 0, the machine produces an output of 1 under input 000, and at time 1, the input is changed to 001. This input results in a change in the value of the output, and since x_3 is the only input variable which is

changed, input x_3 must be fault-free. If pin x_3 were faulty, the internal logic would not have seen an input change at time 1 and could have retained the output $z = 1$. Similarly the output observed at time 4 leads to the conclusion that input x_1 is fault-free, and the output observed at time 7 leads to the conclusion that input x_2 is fault-free. This testing sequence relies upon single input changes which result in output changes and may therefore be applied to machines designed to operate under the single-input-change restriction. Such tests are easy to generate and are quite short.

The following definitions define the flow table property just described.

Definition 4.5: An input line x_i of an asynchronous machine is unit transition testable (UTT) if the flow table for that machine contains two states Q_i and Q_j in input columns I_α and I_β respectively with the following properties: Q_j is a successor of Q_i under the single input change involving input x_i (i.e. I_α and I_β differ only in x_i) and $Z(Q_i, I_\alpha) \neq Z(Q_j, I_\beta)$, where Z is the mapping of total states onto output vectors. The transition from state Q_i to state Q_j is a test transition for input x_i.

Definition 4.6: An asynchronous machine is unit transition diagnosable (UTD) if all of its inputs are UTT.

Fault detection sequences for unit transition diagnosable machines may be designed as follows. Select one test transition for each input line. Form a sequence which contains each of these test transitions by determining

a synchronizing sequence to initiate the test, and appropriate transfer sequences which link the synchronizing state and the test transitions. If the machine is designed assuming single input changes, then the intermediate transfer sequences and synchronizing sequence must be constructed appropriately. The machine, whose flow table is illustrated in Figure 4.5, will be used to illustrate the design of this type of fault detection sequence.

Under input 00, stable state A produces an output of 0. The transitions of interest are those to column 01 and column 10 which result in final states producing outputs of 1. Under both 01 and 10 the final state produces a 0 output. Hence state A under input 00 is not useful for detecting faults. State C in column 00 produces an output of 1, and under input 01 the internal state remains unchanged, but an output of 0 is produced. Therefore this transition is a test transition for input x_2. Under input 10 C goes to D which produces a 1. Therefore neither of the stable states in column 00 may be used to test input x_1. The other stable states are similarly examined, and the result is that the following test transitions for inputs x_1 and x_2 exist (the subscripts denote the input column):

$$x_1: \quad B_{01} \rightarrow A_{11},\ D_{11} \rightarrow B_{01}$$

$$x_2: \quad C_{00} \rightarrow C_{01},\ B_{01} \rightarrow A_{00},$$

$$C_{01} \rightarrow C_{00},\ D_{11} \rightarrow D_{10},\ D_{10} \rightarrow D_{11}$$

The sequence 00,01 is a synchronizing sequence leading to state C_{01}. The sequence is completed by forming any sequence which contains test transitions for x_1 and x_2. If single input changes are required, the following sequence

state	x_1x_2 = 00	01	11	10
A	(A),0	C	(A),0	B
B	A	(B),1	A	(B),0
C	(C),1	(C),0	A	D
D	C	B	(D),0	(D),1

(state), output

Figure 4.5. Asynchronous flow table.

constitutes a fault detection test (the asterisks indicate points where the inputs are tested).

input (x_1,x_2):	00	01	00	10	11	01
State:	$\binom{A}{C}$	C	C	D	D	B
Output:	$\binom{0}{1}$	0	1	1	0	1
			*		*	
			x_2		x_1	

Since this circuit has one output line, a fault on this line is also detected by the above sequence. Therefore if this machine is implemented such that the feedback lines are not connected through input and output pins, all multiple pin faults are detected with this sequence of length 6.

For the sake of comparison, this machine is considered in [10], and a fault detection sequence is designed under the fault assumption that no fault increases the number of stable states in any column. The resulting sequence required 28 symbols.

In the event that the feedback lines are potential fault sites, it is necessary to verify that the internal state variables are functioning. Since the above sequence will detect any faulty inputs, it is not necessary to test every total state in order to assure that every internal state is present. It is sufficient to test for one total state in each flow table row. For example, in Figure 4.5, internal states A and C may be tested by checking for the presence of two stable states in input column 00, and internal states B and D may be tested by checking for the presence of two

stable states in input column 10. In the sequence which was designed to test input errors, stable state C under input 00 and stable state D under input 10 are already tested. The remaining internal states may be tested by appending to the above sequence the following suffix:

input (x_1,x_2):	00	10
state:	A	B
output:	0	0

Note that in the fault detection sequence derived, it is unnecessary to test specific transitions. This results from the fact that for asynchronous machines, which respond to input changes, an input line may be completely tested by application of one input symbol. A further ramification of this fact is that it does not matter whether internal states are verified before or after inputs are verified, and further, the problem of using only previously checked transfer sequences does not arise. It is also observed that for this particular example, the sequence of eight symbols which verifies inputs and internal states could have been used to replace the 12 symbol portion of the sequence designed in [10] which is used to verify the existence of the correct number of stable states in each column.

4.3.2. Fault Detection Sequences for Non-Unit Transition Diagnosable Machines

Any nondegenerate flow table must possess some UTT inputs, however it is not necessary that all inputs have the UTT property. If a machine has some non-UTT inputs, then the method described in Section 4.3.1 can not be used to completely diagnose the circuit. Figure 4.6 illustrates a non-UTD

State \ inputs x_1x_2	00	01	11	10
A	Ⓐ,0	C	Ⓐ,1	B
B	A	Ⓑ,0	A	Ⓑ,1
C	Ⓒ,0	Ⓒ,0	A	D
D	C	B	Ⓓ,0	Ⓓ,0

Figure 4.6. A non-UTD flow table.

flow table which was obtained from the flow table of Figure 4.5 by changing some of the output assignments. In this flow table it may be seen that no transition involving input x_2 (assuming that only single input changes are allowed) results in a transition for which output value changes, hence x_2 is not UTT.

The first step in the design of fault detection experiments for non-UTD machines is the identification of the set of inputs which are UTT. The first part of the test sequence will consist of a sequence which tests these inputs, and this sequence is designed in the same way as test sequences for UTD machines. The second part of the test sequence is designed to test the remaining inputs. At the conclusion of the first part of the fault detection sequence (assuming that the circuit under test has produced the correct output sequence), all UTT inputs will be known to be fault-free.

The effects which faults on the untested inputs can have on the behavior of the circuit, given that the tested subset of inputs is fault-free, can be analyzed by examining the total state transition graph. Each state is grouped with the other states which share the same input column, and these groups are arranged graphically so that an ℓ-dimensional space is formed (ℓ is the number of input lines). All arcs in this transition graph which are along dimensions of the space that correspond to tested inputs are known to be present. Arcs along the other dimensions are present only if these input lines are fault-free which has not been determined yet. It is assumed that these lines are faulty until proven otherwise, however it is not known whether these lines are stuck-at-0 or stuck-at-1. If the state transition graph is visualized with these arcs

absent, it can be seen that if q lines are UTT, then the absence of transition arcs along the ℓ-q untested dimensions results in $2^{\ell-q}$ submachines, each contained in a q-subspace of the ℓ-dimensional input space. For example if a 3 input machine has 1 UTT input, x_1, the transition graph appears as shown in Figure 4.7. In this figure, the nodes represent groups of total states, and the edges represent all transitions between total states in the input columns of the adjacent nodes. The solid edges represent transitions known to be present (by the first part of the fault detection sequence), and the dotted edges represent transitions along dimensions corresponding to suspected inputs. Now if inputs x_2 and x_3 are faulty, say $x_2/0$, $x_3/0$, then the machine will behave like the submachine consisting of input columns 000 and 100 and a transition will occur every time the value of x_1 is changed regardless of the values on x_2 and x_3. This submachine is labeled SM_α. Other faults involving x_2 and x_3 restrain sequential behavior to the other submachines consisting of nodes connected by solid edges. If some part of the test sequence determines that x_2 is functioning properly, then submachines consisting of the planes $x_3 = 0$ and $x_3 = 1$ could define the sequential behavior of the faulty circuit.

In order to test non-UTT inputs, it is sufficient to show that for each such input, two submachines which are adjacent in this variable are connected. The example of Figure 4.6 will be used to illustrate this strategy. Figure 4.8 shows the state transition graph for this machine. The sequence 01,11 is sufficient to test x_1 since an output change must occur. Notice however that if fault $x_2/0$ is present and input 01 takes the faulty machine to state A under input 00, the faulty machine will produce

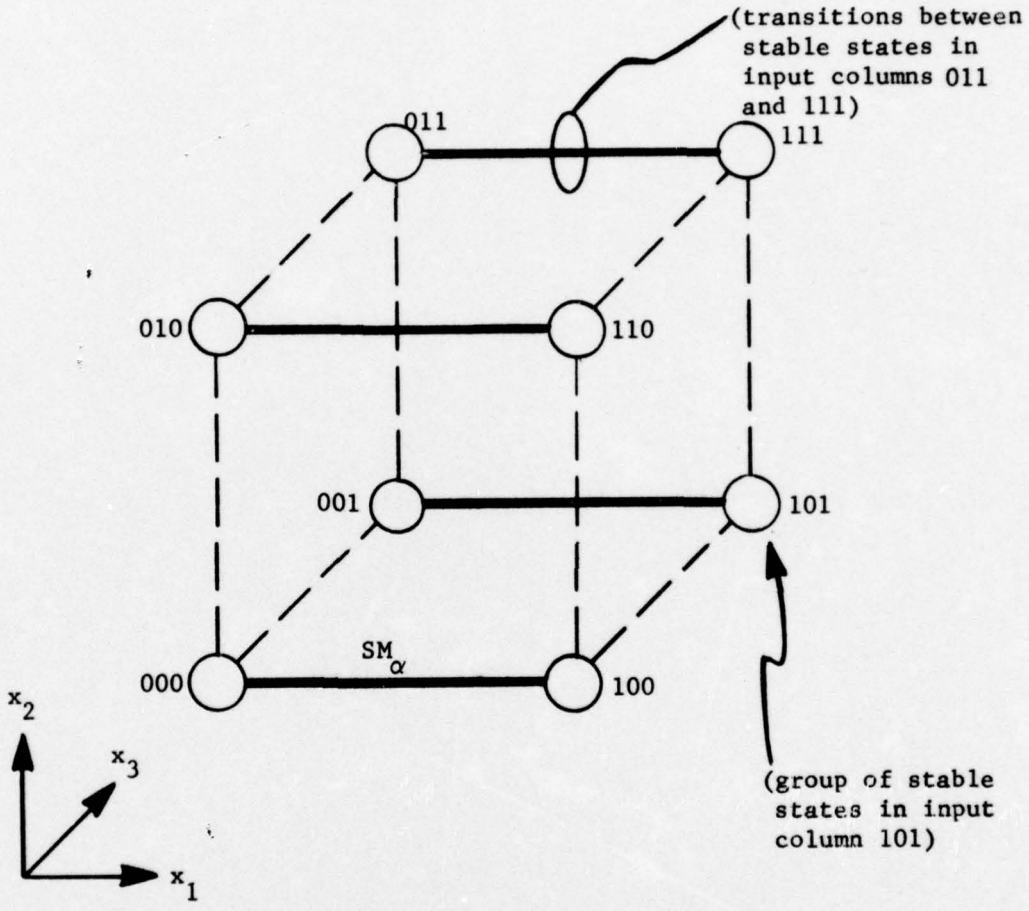


Figure 4.7. Transition graph for a 3 input machine in which x_1 is UTT.

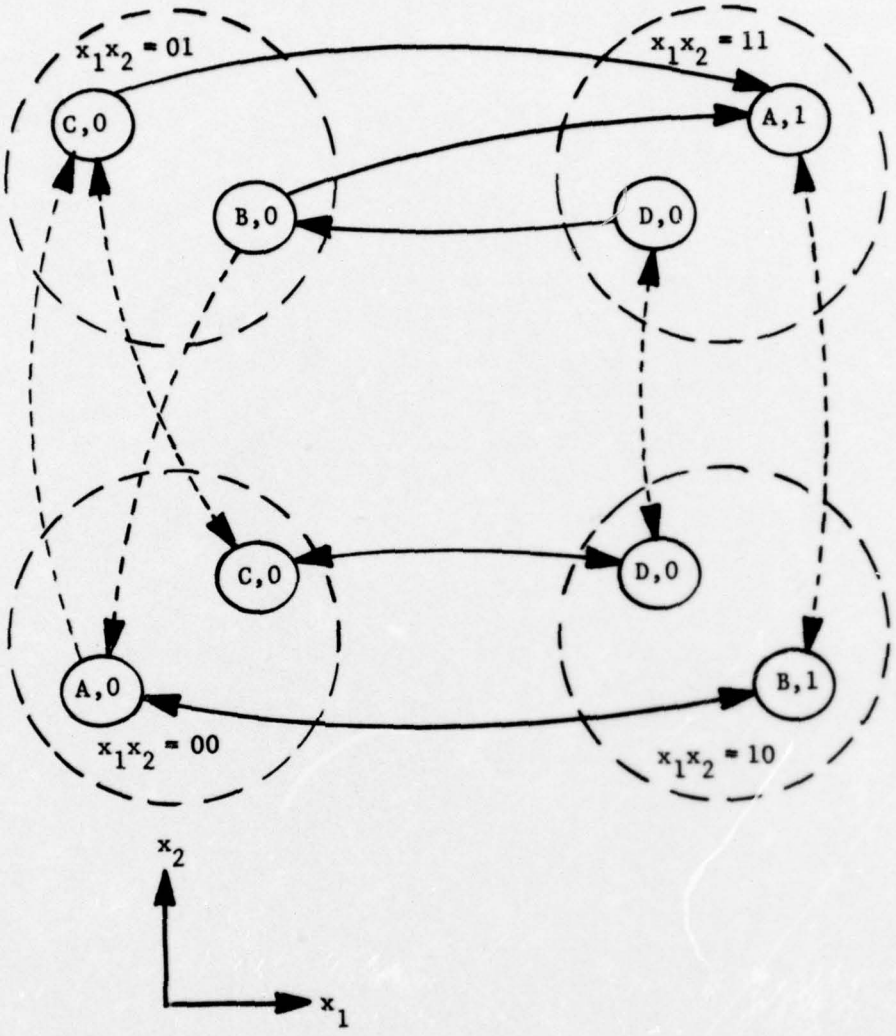


Figure 4.8. State transition graph for machine in Figure 4.6.

the proper response to this sequence. In order to verify that x_2 is fault-free, it is necessary to make a transition along the x_2 dimension which results in the fault-free machine entering a total state which can be distinguished from the state prior to the transition, since if x_2 is faulty no such transition will occur. It is also necessary that the state entered be a state which is distinguishable from any faulty state which the machine could be in after responding correctly to the first part of the experiment, for example state A under input 00 in the present example. A transition from state C under input 01 to state C under input 00 satisfies these conditions. The resulting fault detection sequence is as follows:

input:	01	11	01	00	10
state:	$\binom{C}{B}$	A	C	C	D
output:	0	1	0	0	0

The distinguishability criteria required for forming test sequences for non-UTT inputs are assured by the assumptions that all inputs are irredundant and that the flow table is reduced under the single input change restriction. It is possible, however, that the sequence required to indicate the connectivity of a pair of submachines may require more than one transition in the variable being tested. An example of such a "pathological case" is a machine in which all of the submachines formed by transitions along tested inputs are isomorphic. In this case, it may be possible to transfer from a state in one submachine to a nonequivalent state in the other which solves the problem. However if no such transitions

exist it is necessary to transfer back and forth between these submachines until nonequivalent states are reached by the faulty and fault-free machines.

If the internal state variables require verification, once the input lines are tested, a sequence which tests the feedback lines may be appended. This sequence is designed in exactly the same way as in the UTD case.

5. CONCLUDING REMARKS AND SUMMARY

The research reported in this thesis has been directed at the investigation of a new model, the pin fault model, for failures in digital systems. The goal of this initial study has been to evaluate this model from the standpoint of fault detection test design and to determine the degree of simplification which results. As was discussed in Section 1, an essential characteristic of a fault model is adequate coverage of the various failure modes encountered in the check-out phase and field life of a digital system. A study of dominant failure mechanisms in typical state-of-the-art systems suggests that a fault model of the type proposed here does in fact provide such coverage. It is recognized that additional research is required to establish the validity of the pin fault model, however the required data for such a study is not readily available. The philosophy behind this research is that the model appears to be valid in at least some situations and an investigation of the model from a theoretical viewpoint is therefore justified. In this section the results of the research are summarized, and some observations regarding design of circuits to enhance their testability are made. Areas for additional research are discussed.

5.1. Summary of Thesis

In Chapter 2, the rationale for the pin fault assumption was stated. This rationale is the belief that the dominant failure mechamisms

in digital systems implemented using integrated circuits result in logical behavior modeled by the pin fault approach. It was shown that a substantial reduction in the number of modeled faults results from the pin fault model compared to the stuck-line model. This reduction is based on the typically high ratio of internal lines to the number of input and output pins which characterizes integrated circuits.

Chapter 3 studied the fault detection problem for combinational circuits under the pin fault assumption. A test set of size bounded by 2ℓ was shown to exist for all single output combinational modules. Such test sets are capable of detecting all multiple faults because each single fault is detected "completely." Such test sets also have the property that if applied in the proper order, the normal network response is a strictly alternating sequence. This alleviates the need to store the sequence of normal responses, or to compare the response of the unit under test to a known good unit. The computational aspects of generating such test sets were examined and the results for single output circuits were extended to the multiple output case.

Chapter 4 investigated fault detection in sequential circuits under the pin fault assumption. The synchronous and asynchronous cases were considered individually. Methods for the design of input sequences capable of detecting all multiple pin faults were developed for both synchronous and asynchronous circuits. It was shown that such sequences are much easier to design, and are much shorter under the pin fault assumption than under previous fault assumptions. Fault detection in asynchronous circuits has proven extremely difficult under previous fault

models, however such circuits lend themselves very well to fault detection under the pin fault model, and in many cases may be much more easily diagnosed than in the synchronous case.

The results of Chapters 3 and 4 indicate that the pin fault model provides a more practical approach to fault diagnosis than the stuck-line fault model when large digital systems are considered. A major advantage of this model is that it provides a multiple fault approach without presenting the difficulties encountered when the stuck-line model is extended to the multiple fault case.

5.2. Design Rules to Enhance Testability

In the event that the pin fault model is to be adopted for the design of diagnostic procedures, there may be ways to enhance testability. It is not uncommon for design specifications for various networks to be incompletely specified. This may result because certain combinations of inputs are excluded by constraints which are known to exist among network inputs, or because the response of the network under certain inputs is irrelevant. In either case, the choice of these degrees of freedoms may allow simplified testing.

In the combinational case, the existence of an isolated maxterm or minterm allows the function to be tested with $\ell+1$ trivially generated tests (for a single-output ℓ-input network). If such a function is unspecified for a group of neighboring vertices then assignment of appropriate values to these vertices may be made such that an isolated

min- or maxterm results. If it is not possible to assign unspecified vertices to the ON- and OFF-sets so as to create isolated min- or maxterms, then an assignment which results in minimum size prime implicants or prime implicates will enhance testability.

In the case of sequential circuits, the existence of at least one distinguishing sequence is essential to simple diagnosability. Synchronizing sequences are also desirable, and this function is most conveniently provided by designing such circuits with a reset input. For synchronous circuits, the design of fault detection sequences is facilitated by flow table rows which possess one entry which differs from the rest in either next state or output assignment. Unspecified entries should therefore be selected to provide such rows when possible. Unspecified flow table entries for asynchronous circuits should be selected to render as many inputs unit-transition testable as possible.

Since integrated circuits are packaged in standardized packages, it is not uncommon for unused pins to be available. It is possible to take advantage of this fact to modify the integrated circuit device so that the testing of inputs becomes trivial. To see how this may be accomplished, consider the following function: $Z = x_1x_2\ldots x_\ell \vee \bar{x}_1\bar{x}_2\ldots\bar{x}_\ell$. The three tests $\{t_1 = 1\ 1\ \ldots\ 1, t_2 = 0\ 0\ \ldots\ 0, t_3\}$ where t_3 is any false vertex may be shown to constitute a test set as follows:

$$Z_{UT}(t_1) = 1 \rightarrow P_1E \vee P_2G(11\ldots1)$$
$$\rightarrow \left(\begin{smallmatrix}1 & 1 & & 1\\ n & n & \cdots & n\end{smallmatrix}\right) \vee (00\ldots0)$$

$$Z_{UT}(t_2 = 1) \rightarrow P_2 E \vee P_1 G(0\ 0 \ldots 0)$$

$$\rightarrow \binom{0}{n}\binom{0}{n} \cdots \binom{0}{n} \vee (1\ 1 \ldots 1)$$

$$[Z_{UT}(t_1) = 1] \wedge [Z_{UT}(t_2) = 1]$$

$$\rightarrow [\binom{1}{n}\binom{1}{n} \cdots \binom{1}{n} \vee (0\ 0 \ldots 0)] \cap [\binom{0}{n}\binom{0}{n} \cdots \binom{0}{n} \vee (1\ 1 \ldots 1)]$$

$$\rightarrow (n\ n \ldots n) \vee (1\ 1 \ldots 1) \vee (0\ 0 \ldots 0).$$

The two remaining faults (1 1 ... 1) and (0 0 ... 0) will be detected by t_3. If a realization of this function (or any function comprised of two complementary min- or maxterms) can be added to the integrated circuit chip using an available package pin for the output, then all inputs of the device can be tested with three tests. Figure 5.1 illustrates this circuit modification. It does not matter whether the circuit function is combinational or sequential. A complete test set for such a modified circuit consists of the three tests given above plus inputs which exercise all output pins. The number of additional tests required to exercise the output pins may be reduced by the judicious choice of t_3. As a trivial example, consider a single output combinational function Z which is modified as shown in Figure 5.1. If $Z(t_1) = Z(t_2)$ then t_3 is selected to be one of the vertices (which must exist) such that $Z(t_3) \neq Z(t_1)$. If $Z(t_1) \neq Z(t_2)$ then choice of t_3 is arbitrary. In either case no additional tests are required to exercise the output. The choice of the special diagnosing function may also be used to reduce the number of tests required to test outputs. Any function of either of the following forms is adequate ($\alpha_i \epsilon \{0,1\}$):

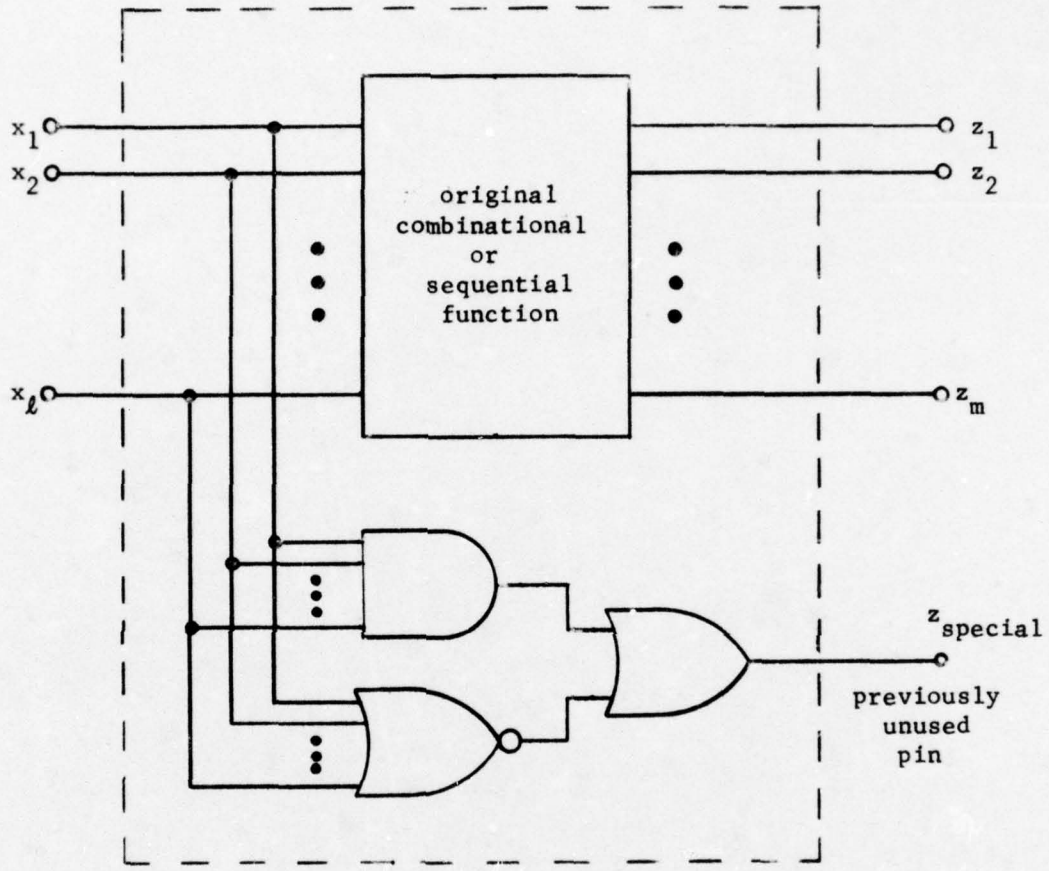


Figure 5.1. Use of available pin to implement input diagnosing function.

$$Z_{special} = \alpha_1\alpha_2 \ldots \alpha_\ell \vee \bar{\alpha}_1\bar{\alpha}_2 \ldots \bar{\alpha}_\ell$$

$$Z_{special} = (\alpha_1 \vee \alpha_2 \vee \ldots \vee \alpha_\ell)(\bar{\alpha}_1 \vee \bar{\alpha}_2 \vee \ldots \vee \bar{\alpha}_\ell).$$

This method of enhancing diagnosability is very attractive because at the expense of only one package pin, diagnosis of an arbitrary number of inputs is achieved with only three tests, and the method is applicable to arbitrary functions, combinational or sequential.

5.3. Areas for Additional Research

The ultimate goal of a fault detection strategy is the capability to test system components at the level of easily accessible subassemblies such as printed circuit boards. Such subassemblies contain many integrated circuits, and therefore fault detection under the pin fault assumption should be investigated for networks of integrated circuit modules. This is a very difficult problem in the general case where arbitrary topologies are allowed. Perhaps certain topological constraints, which must also be reasonable from the logic design standpoint, would result in networks amenable to fault detection.

Minimization of test sets for modules may be critical in the diagnosis of networks of modules. It has been shown that many functions (of ℓ variables) require $\ell+1$ tests, and that some may be diagnosed with fewer. An upper bound on the minimum number of tests required is conjectured to be $\ell+1$, however this bound remains to be established. The proof of this bound may suggest a technique for generating such a test set.

LIST OF REFERENCES

1. J. F. Poage, "Derivation of Optimum Tests to Detect Faults in Combinational Circuits," Mathematical Theory of Automata, Polytechnic Press, Brooklyn, N.Y., 1963.

2. J. P. Roth, et al., "Programmed Algorithms to Compute Tests to Detect and Distinguish between Failures in Logic Circuits," IEEE Trans. on Computers, Vol. C-16, pp. 567-579, Oct. 1967.

3. C. W. Cha, "Multiple Fault Diagnosis in Combinational Networks," Report R-650, Coordinated Science Lab., University of Illinois, Urbana, IL, 1974.

4. W. C. Carter and P. R. Schneider, "Design of Dynamically Checked Computers," IFIP68, Vol. 2, Edinburg, Scotland, pp. 878-883, August 1968.

5. D. A. Anderson, "Design of Self-Checking Digital Networks Using Coding Techniques," Report R-527, Coordinated Science Lab., University of Illinois, Urbana, IL, Sept. 1971.

6. A. Avizienis, "Design of Fault Tolerant Computers," Proc. of Fall Joint Computer Conf., pp. 733-743, 1967.

7. J. von Neumann, "Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components," Annals of Mathematical Studies, No. 34, pp. 43-98, Princeton University Press, Princeton, N.J., 1956.

8. J. G. Tryon, "Quadded Logic," in Redundancy Techniques for Computing Systems, Wilcox and Mann, eds., pp. 205-228, Spartan Books, Washington, D.C., 1962.

9. H. Y. Chang, E. G. Manning, and G. Metze, Fault Diagnosis of Digital Systems, Wiley-Interscience, New York, 1970.

10. A. D. Friedman and P. R. Menon, Fault Detection in Digital Circuits, Prentice-Hall, New York, 1971.

11. D. C. Bossen and S. J. Hong, "Cause-Effect Analysis for Multiple Fault Detection in Combinational Networks," IEEE Trans. on Computers, pp. 1252-1257, Nov. 1971.

12. J. P. Hayes, "Testing Logic Circuits by Transition Counting," Proc. of Fifth International Symp. on Fault-Tolerant Computing, pp. 215-219, Paris, 1975.

13. J. J. Shedletsky, "A Rationale for the Random Testing of Combinational Digital Circuits," Compcon 75, pp. 5-8, Sept. 1975.

14. J. Losq, "Referenceless Random Testing," Proc. 1976 International Symp. on Fault-Tolerant Computing, pp. 108-113, Pittsburgh, PA, 1976.

15. Reliability Analysis Center, "Microcircuit Failure Rates," IIT Research Institute, Dec. 1971.

16. E. F. Platz, "Solid Logic Technology Computer Circuits - Billion Hour Reliability Data," Microelectronics and Reliability, Vol. 8, pp. 55-59, 1969.

17. J. C. McDonald, "Testing for High Reliability: A Case Study," Computer, Vol. 9, pp. 18-21, 1976.

18. National Semiconductor Corporation, "Digital Integrated Circuit Reliability Report," May 1970.

19. R. P. Batni, J. D. Russell, and C. R. Kime, "An Efficient Algorithm for Finding an Irredundant Set Cover," Journal of the Association for Computing Machinery, Vol. 21, No. 3, pp. 351-355, July 1974.

20. D. E. Knuth, The Art of Computer Programming, Vol. 3, Addison-Wesley, Reading, Mass., 1973.

21. S. J. Hong, R. G. Cain, D. L. Ostapko, "MINI: A Heuristic Approach for Logic Minimization," IBM Journal of Research and Development, Vol. 18, No. 5, pp. 443-458, Sept. 1974.

22. O. H. Ibarra and S. K. Sahni, "Polynomially Complete Fault Detection Problems," IEEE Trans. on Computers, Vol. C-24, pp. 242-249, March 1975.

23. J. P. Hayes. "A Study of Digital Network Structure and its Relation to Fault Diagnosis," Ph.D. thesis, Report R-467, Coordinated Science Lab., University of Illinois, Urbana, IL, May 1970.

24. D. R. Schertz, "On the Representation of Digital Faults,"Report R-418, Coordinated Science Lab., University of Illinois, Urbana, IL, May 1969.

25. J. P. Hayes, "The Fanout Structure of Switching Functions," Journal of the Association for Computing Machinery, Vol. 22, pp. 551-571, Oct. 1975.

26. C. D. Weiss, "Bounds on the Length of Terminal Stuck-Fault Tests," IEEE Trans. on Computers, Vol. C-21, pp. 305-309, March 1972.

27. M. R. Paige, "Generation of Diagnostic Tests Using Prime Implicants," Report R-414, Coordinated Science Lab., University of Illinois, Urbana, IL, May 1969.

28. F. C. Hennie, "Fault Detecting Experiments for Sequential Circuits," Proc. of Fifth Annual Symp. on Switching Circuit Theory and Logical Design, pp. 95-110, 1964.

29. E. P. Hsieh, "Optimal Checking Experiments for Sequential Machines," Ph.D. thesis, Dept. of Electrical Engineering, Columbia University, 1969.

30. I. Kohavi, "Fault Diagnosis of Logical Circuits," Proc. of Tenth Annual Symp. on Switching and Automata Theory, pp. 166-173, 1969.

31. J. F. Poage and E. J. McCluskey, "Derivation of Optimum Test Sequences for Sequential Machines," Proc. of Fifth Annual Symp. on Switching Circuit Theory and Logic Design, pp. 121-132, 1964.

APPENDIX

This appendix contains detailed specification of the algorithms developed in Chapter 3. Many of these algorithms are implemented using list processing techniques. In such algorithms, it is assumed that variable AVAIL contains a pointer to the head of a list of available list nodes. When a node is required during the construction of a list, it is obtained from this list of free nodes, and when a node is removed from a list it is returned to the AVAIL list. The fields of the list nodes are specified by name and these field names are illustrated for each type of list node used. For example if pointer Q points to a node of the type shown in Figure 3.2 then the ELT field of that node is referenced as ELT(Q). The symbol λ indicates an empty link field, and is used to terminate lists.

Algorithm A.1: Reduced cost covering set represented in list structure of Figure 3.2.

P[1:q, 1:ℓ]	prime implicant array
C[1:q]	cost vector
R[1:q]	permutation vector
CS	pointer to covering set list
CSSIZE	# of prime implicants in the covet

```
Begin  COVER
    C ← (0,0,...,0)                    /* compute cost vector */
    do  i = 1 to q
        do  j = 1 to ℓ
            if  P(i,j) = s then        /* cost for P(i,*) = # of s's */
                C(i) ← C(i)+1
```

```
     end
end
do  i = 1 to q                    /* initialize R to the natural order */
    R(i) ← i
end
SORT (1,q+1)                      /* form permutation R so that
                                     C(R(i)) ≤ C(R(j)) if i < j */
do for  j = 1 to ℓ                /* form initial weight vector */
     i ← 1
     do while P(R(i),j) = u
        i ← i+1
     end
     W(j) ← i
end
CSSIZE ← 0
CS ← AVAIL
AVAIL ← LINK(AVAIL)
S ← CS
do until W = (0,0,...,0)
     CSSIZE ← CSSIZE + 1
     RLINK(S) ← AVAIL             /* create a node for new element
                                     of cover */
     AVAIL ← LINK(AVAIL)
     k ← max[W(1),W(2),...,W(ℓ)]
     ELT(RLINK(S)) ← R(k)
     Y ← 0                        /* temp for count of the # of
                                     specified variables */
     Q ← RLINK(S)
     do for i = 1 to ℓ
          if  P(R(k),i) = s ∧ W(i) ≠ 0 then do
               Y ← Y+1
               LINK(Q) ← AVAIL
```

```
                    AVAIL ← LINK(AVAIL)
                    VAR(LINK(Q)) ← i        /* store indices of all variables
                    Q ← LINK(Q)                covered by this element which
                    W(i) ← 0                   have not been covered by previous
                                               one */
                end
            end
            SIZE (RLINK(S)) ← ℓ-Y
            LINK(Q) ← λ                     /* terminate sublist */
            S ← RLINK(S)
        end
        RLINK(S) ← λ                        /* terminate list */
        T ← CS
        CS ← RLINK(CS)
        LINK(T) ← AVAIL
        AVAIL ← T
end  COVER
```

Algorithm A.2: Form permutation corresponding to ascending cost of prime implicants. SORT is a recursive procedure which forms a vector R which contains the indices of vector C arranged in order so that $C(R(i)) \leq C(R(j))$ if $i < j$. ℓ and r are pointers into R anc C which define the subfield to be sorted (r points to the first elements not in the subfield). The sort is a straight two-way merg sort.

```
Begin  SORT (ℓ,r)                       /* ℓ,r are local variables */
    if r-ℓ > 1  then do
        SORT (ℓ,ℓ + ⌈(r-ℓ)/2⌉)
        SORT (ℓ + ⌈(r-ℓ)/2⌉,r)
```

```
k ← ℓ
i ← ℓ
j ← ℓ + ⌈(r-ℓ)/2⌉
do until  k = r
     if C(R(i)) ≤ C(R(j)) then do
          T(k) ← R(i)
          k ← k+1
          i ← i+1
          if i = ℓ + ⌈(r-ℓ)/2⌉ then
               do until  j = r
                    T(k) ← R(j)
                    k ← k+1
                    j ← j+1
               end
          end
          else do
               T(k) ← R(j)
               k ← k+1
               j ← j+1
               if  j = r then
                    do until i = ℓ + ⌈(r-ℓ)/2⌉
                         T(k) ← R(i)
                         k ← k+1
                         i ← i+1
                    end
               end
          end
```

```
                do for  i = ℓ to r-1
                    R(i) ← T(i)
                end
            end
        end   SORT
```

Algorithm 3.4: FALSE VERTEX

```
m                       size of universe
F[1:m]                  solution vector
CUBE(ptr) [1:m]         see Figure A.1
Begin  FALSE VERTEX
    Call  DISJOINT CUBES (ILIST,SIZE)
    do i = 1 to m
        F(i) ← u
    end
    do  j = 1 to m
        F(j) ← 0
        Count ← 0
        Q ← CO
        S ← LINK(CO)
        R ← C1
        do until  S = λ
            if CUBE (S)(j) = 0  then do    /* ∩ ≠ ∅; count vertices */
                temp count ← 1
                do for k = j+1 to m
                    if  CUBE (S)(k) = u then temp count ← temp count *2
                end
                count ← count + temp count
                Q ← S
                S ← LINK (S)
```

```
            end
            else if CUBE(S)(j) = 1 then do
                 LINK(R) ← S
                 R ← LINK(R)
                 LINK(Q) ← LINK(S)
                 S ← LINK(S)
            end
            else do                    /* don't care:  count half and put other half
                 temp count ← 1           in other list */
                 do for k = j+1 to m
                      if CUBE(S)(k) = u then temp count ← temp count *2
                 end
                 LINK(R) ← AVAIL
                 AVAIL ← LINK(AVAIL)
                 CUBE (LINK(R)) ← CUBE (S)
                 R ← LINK(R)
                 Q ← S
                 S ← LINK(S)
            end
       end
            if count = 2 ** (m-j) then do    /* if all vertices are true then
                 F(j) = 1                       change to other subtree, and
                 LINK(R) ← λ                    use other list */
                 LINK(Q) ← AVAIL
                 AVAIL ← LINK(CO)
                 CO ← → C1
            end
            else do
                 LINK(R) ← AVAIL            /* throw away other list */
                 AVAIL ← LINK (C1)
            end
       end
  end    FALSE VERTEX (F)                  /* return F to calling proc. */
```

Algorithm A.3: DISJOINT CUBES - called by FALSE VERTEX

m size of space

C pointer to list of cubes (Figure A.1)

CUBE(ptr)[1:m] see Figure A.1

```
Begin  DISJOINT CUBES (C,m)
     S ← C                              /* for each cube in list - */
     do until  S = λ
          Q ← S
          do until LINK(Q) = λ          /* - scan remaining cubes making each
                                           disjoint from CUBE (P) */
               NULL ← 0
               do for i = 1 to m while  NULL = 0
                    if  CUBE (P)(i) = u then I(i) ← CUBE(LINK(Q))(i)
                    else if  CUBE(P)(i) = CUBE(LINK(Q))(i) then
                         I(i) ← CUBE(LINK(Q))(i)
                    else  NULL ← 1
               end
               if  NULL = 0 then do     /* if intersection ≠ ∅ */
                    T1 ← AVAIL          /* T1 pts to new node to be created */
                    do  i = 1 to m      /* create list of disjoint cubes */
                         if (CUBE(LINK(Q))(i) = u) ∧ (I(i)¬= u) then do
                              CUBE (T1) ← CUBE(LINK(Q))
                              CUBE(T1) ← ¬ I(i)
                              T2 ← T1
                              T1 ← LINK (T1)
                         end
```

```
                    end
                    T3 ← LINK(Q)              /* link in new list */
                    LINK(Q) ← AVAIL
                    LINK(T2) ← LINK(T3)
                    AVAIL ← T3                /* place old cube in AVAIL */
                    LINK(T3) ← T1
                    Q ← T2                    /* advance Q to next cube in old
                                                 list */
                end
                else  Q ← LINK (Q)
            end
            S ← LINK(S)
        end
end  DISJOINT CUBES
```

Algorithm 3.5: TEST GEN. TEST GEN operates on array, P, containing q prime implicants of the ℓ-variable function to be tested. COVER is called to construct the list structure illustrated in Figure 3.2. DMATRIX is called to compute the necessary distance information. This information is stored in array D, where D(i,j) contains the distance (up to 2) between prime implicant P(i,*), a member of the covering set, and prime implicant P(j,*). Then for each variable, TEST GEN determines the list of intersections between the prime implicants and the subcube from which a false test vertex must be selected. This list has the structure shown in Figure A.1. The cube field of a node in ILIST is a vector. If pointer S points to a node in ILIST then CUBE (S) denotes the vector stored in this node, and CUBE (S)(i) denotes the i-th component of the vector stored in the cube field of the node pointed to by S. The test set generated is stored in

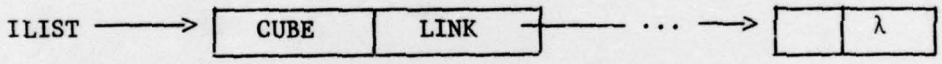


Figure A.1. Structure of intersection list.

three dimensional array T, where T(i,*,0) contains the false test for input pin i, and T(i,*,1) contains the true test for input pin i.

```
Begin  TEST GEN
     call  COVER
     call  DMATRIX
     S ← CS
     do until  S = λ
          Q ← LINK (S)
          do until  Q = λ
               ILIST ← LINK(AVAIL)
               R ← AVAIL
               do for i = 1 to VAR(Q)-1, VAR(Q)+1 to q
                    if D((VAR(Q),i) = 0 ∧ P(i,VAR(Q)) = u then
                         do for j = 1 to ℓ
                              if  P(VAR(S),j) = u then
                                   CUBE(LINK(R))(j) ← P(i,j)
                              end
                              R ← LINK (R)
                         end
                         else if D(VAR(Q),i) = 1 ∧
                                 P(i,VAR(Q)) ≠ P(ELT(S),VAR(Q))
                             then do for j = 1 to ℓ
                                 if  P(ELT(S),j) = u then
                                   CUBE(LINK(R)(j) ← P(i,j)
          end
          LINK(AVAIL) ← LINK(R)
          LINK(R) ← λ
```

```
                    F(*) ← FALSE VERTEX (ILIST,SIZE(S))
                    T(VAR(Q),*,0) ← P(ELT(S),*)
                    i ← 1
                    do j = 1 to ℓ
                        if T(VAR(Q),j,0) = u then do
                            T(VAR(Q),j,0) ← F(i)
                            i ← i+1
                        end
                    end
                    T(VAR(Q),*,1) ← T(VAR(Q),*,0)
                    T(VAR(Q),VAR(Q),1) ← ¬ T(VAR(Q),VAR(Q),1)
                    Q ← LINK(Q)
                end
            end
        end   TEST GEN
```

Algorithm 3.5: DMATRIX. DMATRIX computes the distance between members of the covering set (pointed to by CS) and the other prime implicants.

```
Begin  DMATRIX
    S ← CS
    do for  i = 1 to CSSIZE
        do for  j = 1 to q
            D(i,j) ← 0
            do for k = 1 to ℓ until D(i,j) = 2
                if (P(ELT,k) = 0 ∧ P(j,k) = 1) ∨ (P(ELT,k) = 1 ∧ P(j,k) = 0)
                then  D(i,j) ← D(i,j)+1
            end
        end
        S ← RLINK(S)
    end
end  DMATRIX
```

VITA

Mark Loren Ketelsen was born in Ames, Iowa on November 5, 1946. He received the B.S. degree in electrical engineering from Iowa State University, Ames, Iowa in 1969. He was employed by Texas Instruments, Incorporated from 1969 to 1970, where he was involved in the design of airborne computer systems. From 1972 to 1974 Mr. Ketelsen was a research assistant at the Digital Computer Laboratory, and from 1974 to 1976 he was a research assistant in the Digital Systems Group of the Coordinated Science Laboratory. He received the M.S. degree in electrical engineering in 1973.

Mr. Ketelsen is a member of Tau Beta Pi, Eta Kappa Nu, and Phi Kappa Phi, and was a University Fellow in 1970 and 1971.